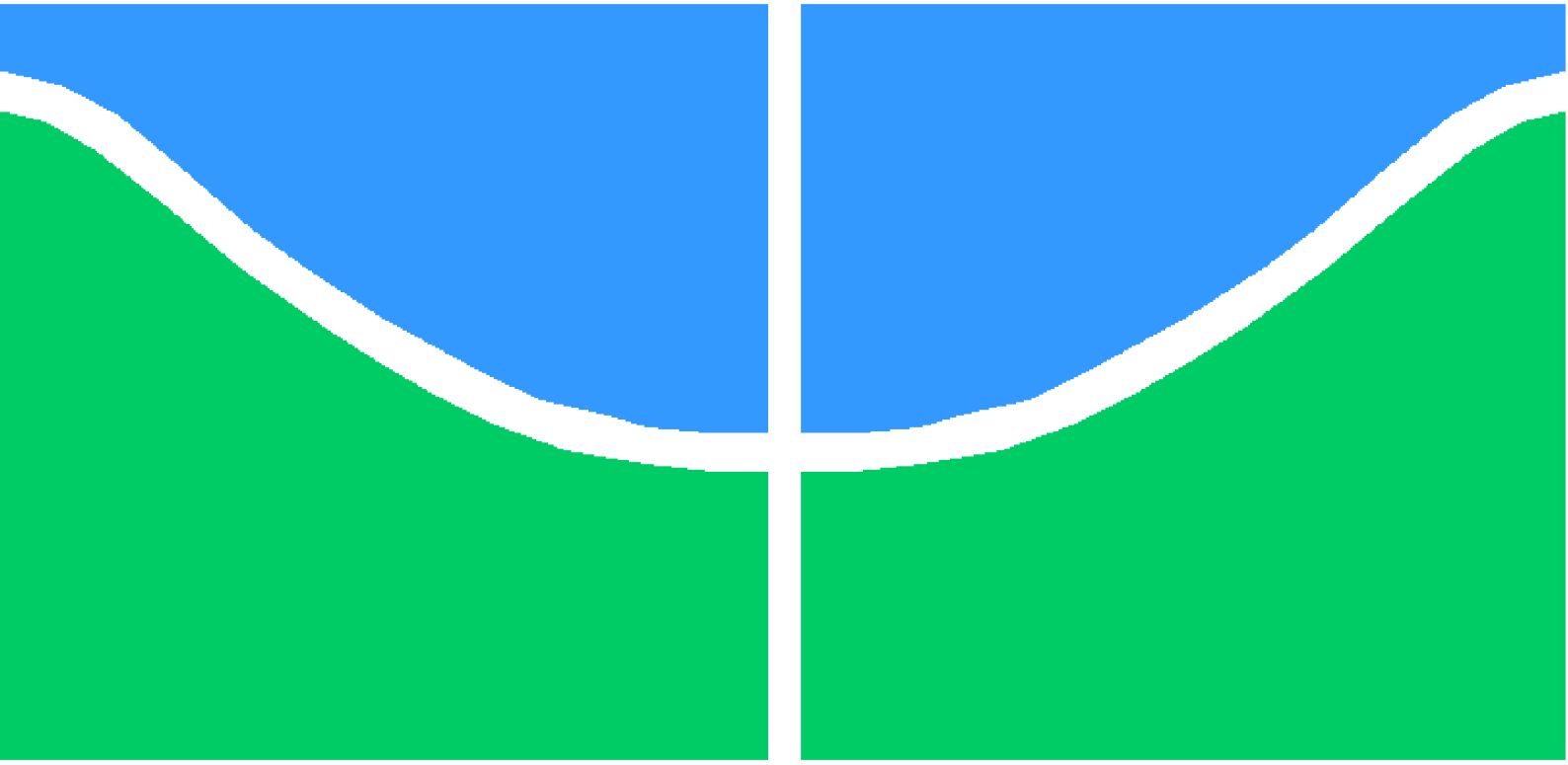

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Estimativa da qualidade de mapas procedurais para jogos do gênero *roguelike*

Autor: Gustavo Jaruga Cruz
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2014

Gustavo Jaruga Cruz

Estimativa da qualidade de mapas procedurais para jogos do gênero *roguelike*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2014

Gustavo Jaruga Cruz

Estimativa da qualidade de mapas procedurais para jogos do gênero *roguelike*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 01 de junho de 2013:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Profa. Dra. Carla Denise Castanho
Convidado 1

Profa. Dra. Carla Silva Rocha Aguiar
Convidado 2

Brasília, DF
2014

Resumo

Conteúdos gerados proceduralmente, isto é, criados de maneira automática e aleatória, estão cada vez mais presentes em jogos para dar o apelo a sua rejogabilidade ao deparar-se com situações sempre diferentes. A adequação destes mapas automatizados pode por vezes se basear apenas no sentimento sem nenhuma métrica efetiva para aferir a qualidade dos mesmos o que pode levar o processo da construção dos algoritmos de mapas longo e repetitivo.

A fim de identificar métricas que caracterizem para um bom mapa foram analisadas as principais características da jogabilidade do gênero *roguelike*. Tais métricas caracterizarão um indicador que será utilizado para estimar a qualidade de um mapa. Para tal fim, foi desenvolvida uma ferramenta capaz de realizar testes através de jogos simulados automáticos para um dado mapa de entrada, que permitiu a extração das métricas de jogo e a estimação da qualidade do mapa para assegurar que os valores esperados para o mapa estão correspondentes as expectativas.

O objetivo deste trabalho não é retirar o usuário do processo e sim deixar nas mãos dos desenvolvedores uma poderosa ferramenta capaz de agilizar o processo de produção de mapas e aumentando a eficácia dos testes utilizando usuários.

Palavras-chaves: mapas. *roguelike*. procedural. qualidade.

Abstract

Procedurally generated content, that is, created in an automated and random manner, are increasingly present in games to give the appeal for its replayability by facing different situations each time. The adaptation of such maps can be at times based on feeling without any effective metric to support it. That can lead to long and repetitive tests.

In order to identify the metrics that defines a good map, the main characteristics of the roguelike genre was analysed. Such metrics will compose a indicator which will be used to estimate the quality of a map. It was developed a tool capable of using a map data as input to perform an automated simulation of games to assure the quality of the map based in several metrics and their expected values.

The objective of this work is not to remove the user from the proccess but to give the developers a powerfull tool capable of quicken the production of automated maps and increasing the effectiveness of the user in the latter tests.

Key-words: maps. roguelike. procedural. quality

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
IA	Inteligência Artificial
SDL	*Simple Direct Layer*
SFML	*Simple Fast Media Layer*
Tiles	Blocos
RPG	*Role-Playing Game*
IDE	*Integrated Development Environment*

Sumário

Introdução

Introdução

É cada vez mais visível em jogos a utilização de conteúdos proceduralmente gerados, isto é, conteúdos gerados automaticamente, de forma aleatória (ou pseudo-aleatória). A demanda por jogos com maiores níveis de rejogabilidade aumentou muito nos últimos anos (HENDRIKX SEBASTIAAN MEIJER, 2011), o que é observado através da alta utilização de conteúdos procedurais em jogos modernos, com gráficos de perspectiva tanto 2D quanto 3D. Alguns exemplos de jogos populares que utilizam conteúdos procedurais podem ser visto em *minecraft*, *terraria*, *starbound*, que possuem seus mapas todos proceduralmente gerados, outros exemplos podem ser vistos também em jogos como *borderlands*, que apesar de possuir mapas regulares, possui a geração procedural fortemente presente na criação de seus itens.

Não se pode esquecer a influência dos primeiros jogos que utilizaram este tipo de conteúdo gerado automaticamente. O gênero *roguelike* surgiu em meados dos anos 80 no jogo *Rogue*, que popularizou-se na época devido a seu aparente infinito número de possibilidades de jogabilidade, consequência do fato de que grande parte do seu conteúdo, como mapas e itens, serem randômicamente gerados, assim como seus gráficos em ASCII (*American Standard Code for Information Interchange*).

Com a evolução dos computadores e dos jogos em si, jogos *roguelike* tiveram que evoluir também, apesar de ainda haver jogos recentes que preservam o aspecto clássico do gênero, com gráficos ainda em ASCII ou formados por *tiles* 2D. Existem também jogos do gênero com gráficos mais modernos e animações, em plataformas móveis, como *Pokemon Mystery Dungeon*[1] e *Izuna*[2], de Nintendo DS e em plataformas recentes como *Guided Fate Paradox*[3], para PS3, o que mostra que o gênero ainda tem público nos dias atuais. Apesar de usarem fortemente conteúdos gerados proceduralmente, ainda hoje não foi estabelecida uma métrica formal para estimar a qualidade destes conteúdos gerados proceduralmente, em especial os mapas, sendo a adequação e validação dos mapas gerados auferidas através de muitos testes e do sentimento obtido após muitas partidas.

Embora os jogos não exijam rigor científico e formal em sua concepção e codificação, a existência de uma ferramenta capaz de produzir métricas de expectativas de qualidade para os mapas randomicamente gerados, de forma automática e rápida, aceleraria o processo de aperfeiçoamento dos algoritmos, graças à redução do tempo devotado

[1] Pokemon - (CHUNSOFT, 2013a)
[2] Izuna - (CHUNSOFT, 2013b)
[3] Guided Fate to Paradox - http://nisamerica.com/games/guided_fate_paradox

aos testes.

Neste trabalho, foi primeiro analisado as diversas características do gênero *roguelike* para que se conseguisse reproduzir em simulação as suas principais características com um certo nível de fidelidade. Foram também testadas diversas formas de se obter um índice de qualidade para um mapa através de várias métricas obtidas sobre jogos realizados.

Para a extração, obtenção e análise das métricas foi construído uma ferramenta capaz realizar simulações automatizadas de jogos sobre um mapa de entrada e através das métricas colhidas estabelecer uma forma de falar o quão bem o mapa se encaixa nas expectativas do usuário, produzindo também como saídas todas as métricas e seus valores obtidos. A outra ferramenta produzida pode realizar a construção de mapas pelo usuário, permitindo a criação de inimigos e itens. Ela também fornece uma interface de construção de *scripts* Lua capaz de mostrar o resultado produzido visualmente em tempo real.

Por fim foram realizados experimentos utilizando diversos mapas e *scripts* procedurais para verificar tanto a qualidade das métricas coletadas em comparação com jogos automatizados e realizados por humanos quanto para tentar analisar possíveis fraquezas dos algoritmos e a diferença que a alteração dos parâmetros de entradas dos mesmos impactam sobre as métricas e qualidades dos mapas.

Organização do documento

O próximo capítulo irá apresentar a origem e a história do gênero de jogo *roguelike*, assim como uma base teórica sobre conteúdos procedurais e taxonomias baseadas em artigos científicos e notícias.

No Desenvolvimento serão descritas as técnicas e procedimentos utilizados para a concepção da pesquisa e desenvolvimento do trabalho, assim como a metodologia aplicada para a concretização dele.

O penúltimo capítulo, Resultados, irá descrever os resultados obtidos a partir das métricas identificadas e das simulações realizadas. Por fim, serão apresentadas as conclusões e possíveis estudos futuros.

O último capitulo irá descrever a conclusão e contribuições das quais podem ser obtidas pela finalização deste trabalho e das ferramentas produzidas durante a sua produção.

1 Referencial Teórico

Aqui serão apresentados os recursos necessários para auxiliar na leitura e entendimento dos temas trabalhados. Primeiro será apresentado uma descrição sobre geração procedural, suas taxonomias e termos adotados. Serão então mostradas técnicas encontradas sobre a geração de mapas sob perspectiva 2D e por fim uma breve história sobre o gênero *roguelike* e a sua evolução no contexto do mercado.

1.1 Geração Automática Procedural

A utilização de algoritmos e técnicas para a criação de conteúdos é chamada de Geração Procedural. A definição do termo (embora não exista uma versão definitiva e consagrada na literatura) diz respeito à geração automática, aleatória ou pseudo-aleatória, de dados, sejam objetos, figuras, sons ou imagens.

1.1.1 Geração Procedural vs Geração Procedural de Conteúdo

Apesar de não possuir uma literatura oficial, a **Geração Procedural** é dividida por diversas fontes e comunidades (DOULL, 2013) em dois termos de acordo o seu objetivo:

- **Geração Procedural**
  A definição original do termo. Representa dados gerados de forma aleatória ou pseudo-aleatória. Este termo ainda é utilizado de forma genérica para referenciar qualquer forma de geração, porém é comum utilizá-lo para se referir a produtos visuais, sonoros ou objetos que não afetem a jogabilidade.

- **Geração Procedural de Conteúdo**
  O termo refere-se à geração de produtos que afetam a jogabilidade de alguma forma. Aqui encontram-se a geração de terrenos e mapas, itens, objetivos e outros fatores que podem alterar o curso de uma partida.

A distinção dos dois termos surgiu com a diferença óbvia de objetivos e impactos que teriam sobre um jogo. No mercado de jogos atuais, pode-se dizer que quase todos os jogos 3D utilizam geração procedural para a criação de seus mapas, através da disposição randômica de artefatos como árvores, gramas ou texturas ligadas entre si. Porém, estes são elementos que não alteram efetivamente a jogabilidade e o andamento do jogo: sua funcionalidade é de representar mais naturalmente e variadamente atrativos visuais ao usuário, e portanto não caracterizam uma geração procedural de conteúdo.

1.1.2 Taxonomia e Termos

Como dito anteriormente, ainda não há na literatura a definição de termos e formas existentes de geração procedural, porém é visível que algumas técnicas utilizadas são baseadas dos mesmos princípios. Os termos e taxonomias apresentados são propostos por (JULIAN et al., 2011). Entretanto, podem existir variações entre os termos citados em relações a outras fontes. As gerações procedurais podem ser então classificadas de acordo com os seguintes conceitos:

Online versus *Offline*

Indica o modo como o conteúdo é gerado. A forma ***online*** representa que conteúdo é gerado em tempo de execução enquanto na forma ***offline*** o conteúdo é gerado durante o desenvolvimento e inserido no jogo estaticamente.

O método **offline** é muito utilizado para se gerar grandes mapas abertos automaticamente, deixando a cargo do *designer* de níveis o trabalho de modificá-lo manualmente para deixá-los da maneira desejada. Isto permite gerar detalhes mais naturais em mapas muito grande, que de outra forma consumiram uma quantidade inefetiva de tempo para serem produzidos.

O método ***online*** cria conteúdos sem a interferência humana e portanto deve se tomar maiores cuidados com a lógica implementada no algoritmo para garantir que o resultado da geração atenda as expectativas da equipe de desenvolvimento e dos jogadores. É comumente usada para partes não vitais do jogo, onde um erro de lógica não possa resultar em uma impossibilidade de progresso na história, apesar de ser utilizado também de outras formas.

Conteúdo Necessário e Opcional

O **conteúdo necessário** é aquele que o jogador irá interagir para que possa progredir no jogo, enquanto que o **conteúdo opcional** é aquele que o jogador pode optar por experimentar ou não e ainda assim atingir o objetivo principal do jogo.

Sementes aleatórias versus vetores parametrizados

As **sementes aleatórias** e os **vetores parametrizados** representam o modo que os algoritmos procedurais efetuam seus cálculos. Um algoritmo utilizando **sementes aleatórias** pode ser representado pela semente (valor inicial) passada para o seu gerador de números randômicos, enquanto que no outro extremo o algoritmo pode tomar como entrada um **vetor multidimencional de parâmetros** reais para definir mais detalhadamente seus propriedades específicas.

Para um algoritmo de geração de calabouços, alguns dos parâmetros poderiam ser o número de quartos, índice de ramificação dos corredores, o tamanho dos corredores, etc. Para um algoritmo de geração de mapas globais alguns parâmetros poderiam ser a porcentagem de água, a quantidade de ilhas, o clima geral do planeta, o índice de ocorrência de montanhas e minérios, etc.

Geração estocástica versus determinística

Uma **geração determinística** é aquela que, dado os mesmos parâmetros de entrada, produz sempre o mesmo resultado. A **geração estocástica** é aquela que sempre produz resultados distintos, mesmo que as entradas sejam idênticas.

A geração estocástica é muito comum para **algoritmos parametrizados**. Em um gerador de calabouços, dado os mesmo parâmetros de quantidade de salas e tamanho do mapa total, as saídas não resultam em mapas idênticos, apesar de possuírem entradas iguais.

Construtivo versus Gerar-e-testar

Um **algoritmo construtivo** gera seu conteúdo apenas uma vez e o termina. É necessário tomar os devidos cuidados durante a geração para garantir que a saída seja suficientemente aceitável. Um **algoritmo de gerar-e-testar** é aquele que cria o conteúdo e, ao finalizá-lo, realiza um ou mais testes para garantir que a saída atende aos critérios especificados. Caso não atenda, partes do conteúdo são geradas e testadas novamente, até que os critérios de aceitação sejam atingidos.

Os mapas de *Dwarf Fortress* são gerados utilizando vetores parametrizados que descrevem o tamanho, clima e idade do mapa. Durante a criação do mapa é possível visualizar um informativo mostrando a `quantidade de áreas rejeitadas.` A figura 1 mostra um exemplo real do algoritmo de gerar-e-testar.

Figura 1 – Criação de mapas em *Dwarf Fortress*: 93 mapas rejeitados

1.1.3 Técnicas e Algoritmos

Esta seção irá apresentar algumas técnicas e algoritmos utilizados para a geração procedural, tais como a utilização de ruídos para a criação de formações montanhosas e erosões simuladas (DISCOE, 2013). A divisão entre visão topológica e estrutural de mapas utilizando uma bem definida gramática de grafos (ADAMS, 2002). A geração procedurais de sons ambiente (FARNELL, 2010). E técnicas de gerações de mapas 2D, que apesar de não serem encontradas em artigos científicos, podem ser vistos em diversas publicações *online* (TOMPSON, 2011) (NAUGHTYYT, 2013).

1.1.3.1 Ruído

Na geração procedural como um todo, mas principalmente na geração de terrenos 3D, a utilização de algoritmos de ruídos é algo de grande importância, dado o resultado aparentemente mais natural e visivelmente aleatório obtido por eles (GOKTAS, 2013).

Terrenos

Atualmente é a área da geração procedural que possui a maior disponibilidade de informações e técnicas reconhecidas. A geração procedural de terrenos se popularizou muito na última década devido a uma multidão de jogos de mundo aberto com terrenos cada vez mais detalhados e únicos.

Um dos métodos mais populares e utilizados para a geração de terrenos 3D e ou terrenos 2D de plataforma é a criação de fractóides. Terrenos são comumente gerados inicialmente a partir da criação de *heightmaps*, mapas de altura que irão definir a altura do terreno. Um exemplo de algoritmo de criação de *heightmaps* utiliza um processo iterativo sobre **ruído fractal Browniano** (*fractal Brownian noise*) (DISCOE, 2013). Outro exemplo visto é na criação de superfícies sintéticas, algoritmos de geração planetares que utilizam o **ruído de Perlin** (*Perlin noise*) (TULLEKEN, 2009). Outra técnica utilizada é a simulação de erosões sobre o terreno (DISCOE, 2013).

Texturas

A utilização de ruídos é muito comum para a criação de texturas procedurais, como texturas de árvores e terrenos, para mostrar um ambiente mais vivo. Este é um exemplo do termo geração procedural que não envolve conteúdo relevante à jogabilidade.

Sons

A utilização de ruídos e filtros através de algoritmos procedurais pode servir como base para a geração de sons. Um uso mais comum dessa técnica é a geração de sons de

diversos tipos de ventos ou sons ambientes. Uma grande literatura sobre este assunto pode ser observada nos trabalhos de *Farnell* (FARNELL, 2010) (FARNELL, 2013).

1.1.3.2 Geração de terrenos em perspectiva 2D

Informações sobre geração de mapas 2D são mais dificilmente encontradas e há atualmente uma falta de formalização das técnicas utilizadas. Algumas técnicas observadas são:

- **Montagem estática:** Vista principalmente em jogos de plataforma 2D e *dungeon crawlers (2D e 3D)*, esta técnica constitui-se de uma seleção de mapas pré-definidos e armazenados que são "encaixados" uns aos outros durante a criação do mapa, dando ao jogo uma certa diversidade de conteúdo com a segurança de que os mapas gerados serão consistentes com o *design* e as mecânicas do jogo.
- **Graph Grammar:** O trabalho (ADAMS, 2002) mostra a técnica de criação de calabouços utilizando **Gramática de Grafos** (*Graph Grammar*). Em seu documento, ele divide a criação de mapas em duas etapas: **Topologia** e **Geometria.**

  A **topologia** do nível descreve como a ordem das coisas deve acontecer, sem se preocupar com os aspectos físicos ou detalhes dos terrenos. Um exemplo mais simples de um modelo topológico (representado visualmente pela Figura 2) de um nível pode ser:
  `Inicio->Inimigo->Inimigo->Vida->Inimigo->Item ->Chefe->Fim`

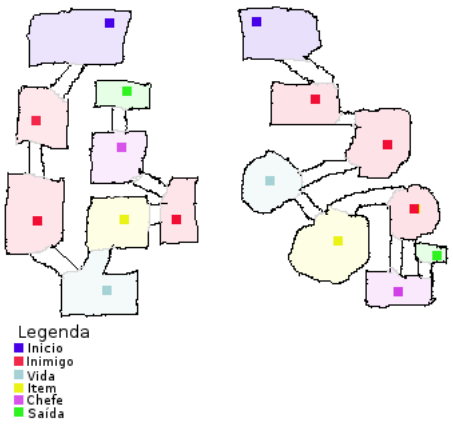

Figura 2 – Diferentes mapas com o mesmo modelo topológico acima.

Um exemplo mais complicado (representado visualmente pela Figura 3), porém menor, poderia ser:
`Inicio->Inimigo->Bifurcação[Bifurcação(Item->Item)->Inimigo]->Chefe->Fim.`

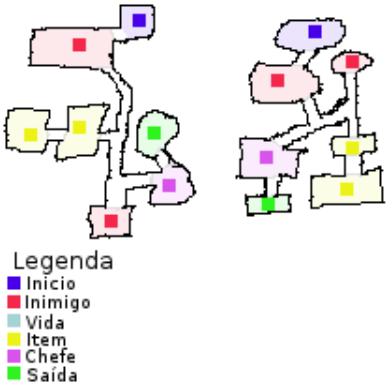

Figura 3 – Diferentes mapas com o mesmo modelo topológico mais complexo acima.

A geração **geométrica** é aquela que utiliza os dados obtidas da topologia e, a partir destas entradas e um certo grau de randomicidade, gera fisicamente as salas e detalhamentos, assim como o posicionamento de inimigos e itens.

Também é visto a geração de conteúdo procedural através de `Gramática de Grafos` utilizando células *voronoids*[1] para a criação de salas e corredores em (NAUGHTYYT, 2013). O autor comenta que sua principal inspiração para a sua criação foi inclusive o trabalho de (ADAMS, 2002).

- **Miners**
  Esta técnica foi apresentada por *Jeffrey Thompson* (TOMPSON, 2011) para a criação de cavernas. A ideia é basicamente criar um mundo constituído apenas por blocos, e colocar células randômicas espalhadas, chamadas de **mineradores** (*miners*). O algoritmo irá entrar entrar em um laço onde selecionará os *miners* ativos e chamará sua função *dig* (cavar), que terá uma pequena chance de gerar outro minerador adjacente a ele e que irá destruir um bloco adjacente, movendo-se até ele. Caso não haja mais nenhum bloco adjacente ao *miner*, ele será desativado. A inserção de outros limites para a desativação de *miners* também pode ser utilizada, como tempo de vida ou a quantidade de mineradores criados por ele.

  Após a desativação de todos os mineradores o algoritmo entrará na fase de limpeza, onde irá remover defeitos, imperfeições obviamente irregulares como "paredes solitárias", "pequenas ilhas" e outras, definidas a critério do *game designer*.

[1] Célula representando uma região de um diagrama Voronoi

1.2 Jogos *Roguelike*

1.2.1 Caracterização do termo *roguelike*

Existem atualmente diversos gêneros e sub-gêneros de categorias de jogos, que constituem uma tentativa de classificá-los de acordo com suas principais características e objetivos. Dentre estes muitos tipos, o gênero *Roguelike* é uma derivação do RPG (*Role-Playing Game*) e *Dungeon Crawl*.

O gênero *Dungeon Crawl* é constituído de um cenário de fantasia onde heróis navegam por labirintos e enfrentam diversos monstros e desafios, encontrando tesouros e recompensas. RPG é o termo que caracteriza jogos onde o jogador interpreta uma personagem em um mundo fantasia. Em jogos este termo está intrinsicamente ligado a progressão de nível e habilidade, e ataques afetadas por parâmetros que podem ser treinados através de pontos de experiência ou outro sistema similar.

O termo *roguelike* é inspirado pelo jogo *Rogue* (Figura 4 - 4), construído para sistemas baseados em Unix em 1980. *Rogue* se destacou na época pela utilização de gráficos ASCII para representar a contextualização de seu jogo e pela disposição das salas e itens, que eram gerados aleatoriamente cada vez que o jogo era jogado, um conceito que poucos jogos utilizavam na época. O jogo se baseava em turnos e o jogador tinha que se mover por labirintos constituídos de 3 por 3 salas dispostas aleatoriamente ao longo do nível.

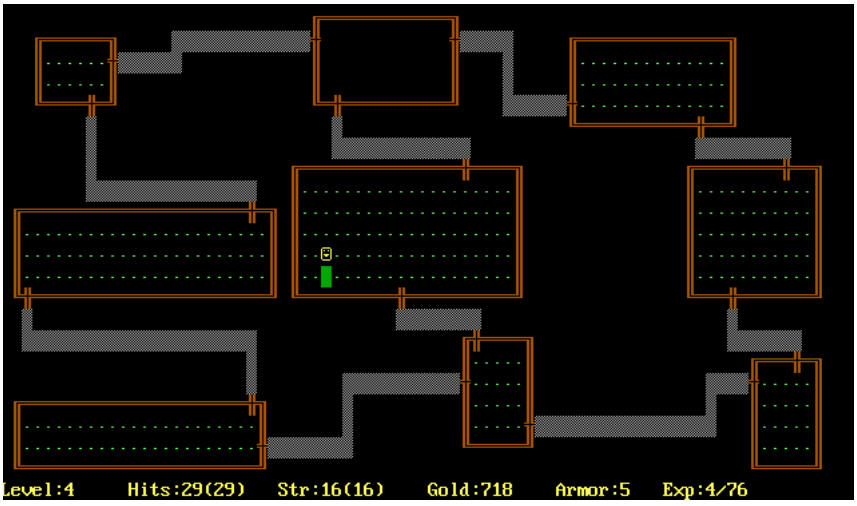

Figura 4 – Rogue em um IBM PC

Rogue sofreu críticas devido ao seu alto custo de CPU para jogos da época, porém a idéia não se desfez e em 1984 o jogo recebeu suas versões para IBM PC e Macintosh. Apenas 2 anos depois do lançamento de *Rogue*, o termo *roguelike* já estava inspirando jogos como *Hack* em 1982 e *Moria* em 1983.

Moria foi talvez um dos mais influentes jogos que ajudaram a desenvolver o termo *roguelike* e torná-lo mais próximo do que é hoje. *Moria* teve sua ambientação inspirada nas

histórias de Tolkien, Senhor dos Anéis (TOLKIEN, 2001), e tinha como objetivo derrotar Balrog nas profundezas das Minas de Mória. *Moria* (Figura 5 - 5) já apresentava mapas aleatórios mais elaborados e maiores, com um sistema mais bem definido de combate e lojas (KOENEKE, 2013).

Figura 5 – Moria em um terminal: '@' representa o jogador, 'r' representa ratos.

O termo *roguelike* é caracterizado pela forte herança de conteúdos procedurais que afetam o jogabilidade, isto é, a forma de se jogar, sejam eles mapas, itens ou a disposição dos inimigos. Apesar de não estar sempre presente, jogos deste gênero possuem o conceito de morte permanente, isto é, uma vez que o seu jogador morra é necessário recomeçar o jogo com uma nova personagem. O gênero também estabelece conceitos como a visibilidade do mapa e movimento baseado em blocos. Uma lista de fatores que característicos do gênero, estabelecendo a sua definição, foram discutidos e compilados durante conferência de desenvolvimento de jogos *roguelike* (*Roguelike Development Conference*), em 2008, e atualizados recentemente (ROGUETEMPLE, 2013).

1.2.2 História

A popularização de jogos *roguelike* foi obscurecida no mercado de jogos ocidental em seu primeiro momento, constituindo-se principalmente de jogos não-comerciais, tendendo a apresentarem gráficos ASCII. Porém, nos anos recentes, é possível notar um crescimento de jogos *roguelike* no mercado de jogos. Os novos jogos do gênero constituem-se principalmente de gráficos animados e bem desenhados, alguns possuindo até mesmo uma ambientação 3D (MOTT, 2013).

Em um primeiro momento após a fixação do termo, jogos comerciais do gênero foram produzidos por marcas orientais como a Chunsoft[2], uma marca japonesa especia-

[2] *Chunsoft* - www.spike-chunsoft.co.jp

lizada em jogos *roguelike*, com títulos como *Shiren The Wanderer*[3] (que mais tarde foi publicado pela Sega no Nintendo DS), assim como *Pokemon Mystery Dungeon*[4] (Figura 6 - 6).

Figura 6 – *Pokemon Mystery Dungeon* - GBA (esquerda) e *Izuna* - Nintendo DS (direita)

Estes novos jogos são, em sua maioria, formados por salas com visibilidade total e corredores obscurecidos, e os jogos usualmente são divididos em duas partes: uma onde o jogador poderá se organizar e comprar novos itens, e pequenas áreas de 5 a 10 andares. O conceito de morte permanente tornou-se menos recorrente: a morte no jogo faz com que o jogador perca todos os itens, experiência e dinheiro adquiridos na área onde foi derrotado.

Esta nova forma de jogo tornou o gênero *roguelike* mais familiar aos jogadores, trazendo diversas traduções para o ocidente e ampliando a divulgação do estilo de jogo. Apesar de ainda não existirem muitos jogos comerciais deste gênero desenvolvidos no ocidente, pode-se observar nos últimos anos um grande aumento de jogos independentes sendo criados. Alguns jogos dentre estes que encontram-se na Steam são *Dungeons of Dreadmor*[5], *Sword of the Stars: The Pit*[6], *Steam Marines*[7], dentre outros (Figura 7 - 7).

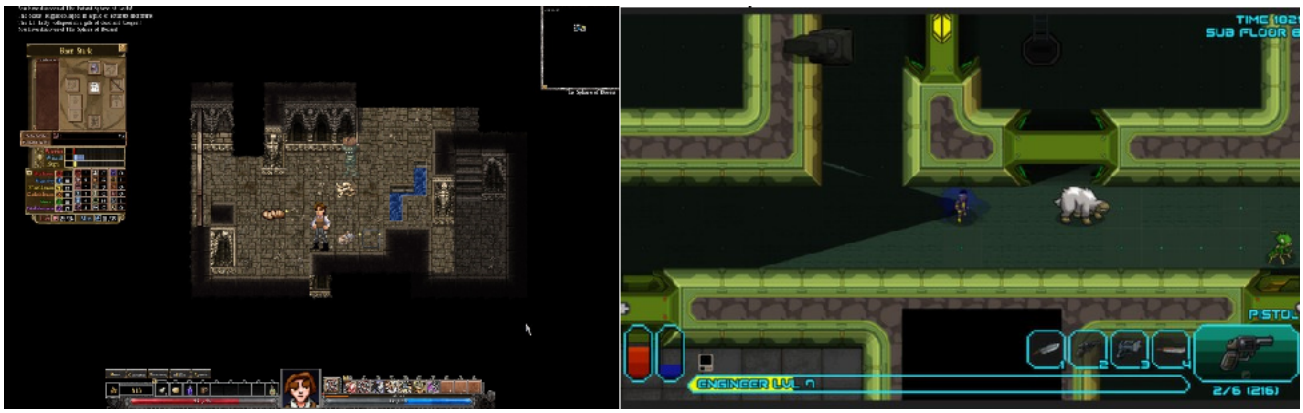

Figura 7 – *Dungeons of Dreadmor* (a esquerda) e *Sword of the Stars: The Pit* (direita)

[3] *Shiren The Wanderer* - http://atlus.com/shiren
[4] *Pokemon* - www.spike-chunsoft.co.jp/games/pokedun/
[5] *Dreadmor* - http://store.steampowered.com/sub/15933
[6] *Sword of the Stars* - http://store.steampowered.com/app/238450
[7] *Steam Marines* - http://store.steampowered.com/app/253630

Vale ainda ressaltar os jogos recentes produzidos pela *Nippon Ichi Software*, que utilizam-se de belos gráficos em um ambiente 3D cheio de animações, o que pode muito bem indicar o futuro de jogos *roguelikes* comerciais. Seus dois jogos: *Zettai Hero Project*[8] (2010) e *Guided Fate to Paradox*[9] (2013), utilizam uma mecânica de movimentação de blocos em turnos em uma perspectiva isométrica, e adiciona também o conceito níveis-Z, isto é, os ataques e movimentos são influenciados pela altura dos terrenos.

Figura 8 – *Guided Fate to Paradox* em sua perspectiva isométrica 3D

[8] *ZHP* - http://nisamerica.com/games/zhp
[9] *Guided Fate to Paradox* - http://nisamerica.com/games/guided_fate_paradox

2 Desenvolvimento

Neste capítulo estarão descritos as ferramentas e técnicas utilizadas para as suas construções, assim como as dificuldades e impasses encontrados durante o desenvolvimento do trabalho.

A principal ferramenta, nomeada *MetaEngine*, foi desenvolvida com o objetivo de realizar uma série de jogos automatizados sobre um mapa, gerados proceduralmente por um algoritmo ou não, e estimar, de acordo com os parâmetros de aceitação do usuário da ferramenta o quão bem o mapa se adequa ao seu esperado.

O primeiro passo para a obtenção de uma medida real e automatizada de métricas foi a construção de uma simulação dentro da aplicação capaz de impor as diversas regras de comportamento do gênero alvo. Para este fim, foi desenvolvida no *MetaEngine* a capacidade de simular um jogo do gênero preservando as suas principais características. Além de ser capaz de simular o jogo, a ferramenta também permite que uma inteligência artificial controle o jogador para a automatização de jogos e viabiliza a extração de métricas das simulações realizadas.

Os diversos aspectos de cada parte da ferramenta serão discutidos em mais detalhes ao longo do capítulo.

Na segunda parte foi o desenvolvimento de uma ferramenta extra, entitulada *MapBuilder*, capaz de criar mapas, tanto de forma manual quanto através de algoritmos procedurais escritos em linguagem Lua[1]. Esta ferramenta serviu para auxiliar na produção de *scripts* proceduais e mapas manuais para serem validados pela ferramenta *MetaEngine*.

2.1 A API Gráfica do Sistema

Devido a facilidade de programação para jogos e prévio conhecimento, a linguagem de programação utilizada para o desenvolvimento da ferramenta foi C++.

Dentro das API's mais conhecidas para programação de jogos em C++, gratuitas, encontram-se:

- Allegro 5,
- SDL 2,
- SFML 2.1.

[1] Linguagem de programação lua - http://www.lua.org/

Allegro é uma API conhecida por sua simplicidade, porém devido a problemas encontrados em sua versão 4 com placas gráficas *onboard*, esta não foi realmente considerada como uma opção viável para o trabalho, dado também o tempo de aprendizado adicional que seria necessário para seu domínio.

A SDL é uma ferramenta que recentemente disponibilizou sua versão 2.0, com grandes mudanças e melhoras de performance, sendo a mais conhecida e utilizada dentre as três API's. É construída C, apesar de garantir suporte nativo a C++.

A SFML é uma API relativamente recente que utiliza-se de diversos benefícios da linguagem C++11 internamente para segurança e integridade. Seus benefícios se dão pela sua construção orientada a objetos, o que facilita na utilização e organização de seus diversos módulos.

Todas as API's descritas são multiplataformas e permitem a compilação de aplicativos para Windows, Linux e MacOS. Um breve comparativo entre SDL e SFML foi realizado, cujos resultados estão compilados na Tabela 1.

Característica	**SDL**	**SFML**
Multiplataforma	Sim	Sim
Orientado a Objetos	Não	Sim
Aceleração de Hardware	Sim	Sim
Integração com Áudio	Sim, através de bibliotecas	Sim
Integração com imagens *.png* e *.jpg*	Sim, através de bibliotecas	Sim
Alto Grau de Maturidade	Sim	Não[2]
Referências e exemplos	Sim	Não[3]
Open Source	Sim	Sim

Tabela 1 – Comparativo entre SDL e SFML

Por fim, optou-se a utilização da SFML devido a suas facilidades, orientação a objetos nativa e objetos primitivos pré-construidos como classes para representação de textos e gráficos.

Foi também utilizado a biblioteca Qt 5[4] para o interfaceamento gráfico, principalmente da ferramenta auxiliar. O Qt disponibiliza diversas características comuns para janelas como menus, botões, e outros padrões gráficos que usuários estão acostumados. A utilização desta biblioteca tornou a ferramenta mais intuitiva e efetiva, enquanto as API's citadas acima, que são muito bem utilizadas para jogos, falham em ter uma padronização de interface, dando a liberdade ao programador de fazê-las, o que para o contexto de um jogo é algo desejável. Por isso foi-se utilizado a SFML para a produção dos gráficos e contextos da simulação e o Qt para a produção da ferramenta.

[4] *Framework* Qt - http://qt-project.org/

2.2 Ferramenta de simulação e análise - *MetaEngine*

Aqui estarão detalhados os diversos módulos da principal ferramenta do trabalho, *MetaEngine*, as suas dificuldades e decisões tomadas durante o seu desenvolvimento.

O *MetaEngine* é dividido em três módulos principais (Figura 9), a Simulação, a Automatização e as Métricas. O módulo de simulação contém estruturas para guardar informações sobre mapas e implementações sobre as regras e interações dos jogos. O módulo de automatização é realizado através da classe *LuaManager*, responsável por mapear as classes e estruturas utilizadas pela ferramenta em C++ para o *Lua*, podendo então carregar as inteligências artificiais e os *scripts* de mapas procedurais. O módulo das métricas é responsável por colher métricas extraídas do módulo de simulação e realizar um processamento para aferir a qualidade esperada de um dado mapa de entrada através das seus métricas obtidas como saída.

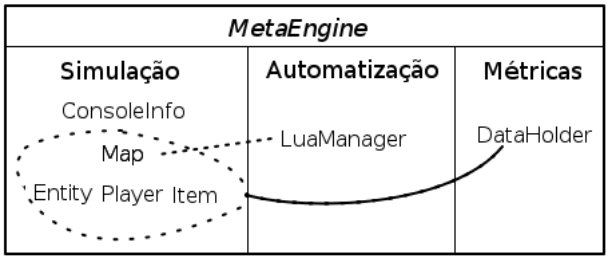

Figura 9 – *MetaEngine* - Divisão de seus módulos

2.2.1 Simulação

Foi construído um ambiente de simulação controlado para os testes que devem ser executados sobre o mapa. O jogo, apesar de simplificado, contém os elementos principais de um jogo *roguelike* que permitam a realização de um comparativo da efetividade de cada mapa para este gênero.

A simulação do jogo presente no *MetaEngine* consta de regras básicas que permitam testar, da forma mais genérica possível, jogos do gênero *roguelike*. O jogador iniciará em um nível carregado a partir de um arquivo externo ou gerado por um algoritmo procedural lido pela ferramenta. O nível carregado deverá ter, mínimamente, um ponto de entrada e um ponto de saída, podendo também haver uma série de inimigos e itens de diferentes atributos espalhados pelo mapa.

Para facilitar o entendimento das regras do jogo, os itens estarão limitados a itens de recuperação, que aumentarão a quantidade de vida do jogador, e itens de atributos, que aumentarão a quantidade de ataque ou defesa do jogador até o término do nível, se coletados.

2.2.1.1 Turnos

A simulação se desenvolve em turnos. Um turno só ocorrerá caso o jogador faça um movimento, que pode ser um movimento a um espaço vazio ao seu redor, ou um ataque a um inimigo adjacente. Ao realizar um turno, todos os outros inimigos do mapa irão também realizar seus turnos e mover-se em direção ao jogador para atacá-lo, caso estejam em sua área de observação.

Uma visão mais detalhada do desenvolvimento dos turnos pode ser encontrada no Apêndice A.

2.2.1.2 Mapas

O mapa é dividido em *tiles* (blocos). Cada bloco é representado por um quadrado que pode ser passável ou não. Em cada bloco pode haver até um item ou um inimigo ao mesmo tempo. Ao passarem por cima de itens, os jogadores consumirão para aumentar seus atributos ou recuperar sua vida. Jogadores e inimigos não podem coexistir em um mesmo bloco, havendo um ataque caso um inimigo ou jogador tente se mover para um bloco ocupado.

A ferramenta terá a capacidade de leitura de um arquivo externo *.map*, que é que um arquivo de texto com a devida formatação entendida pela ferramenta. O arquivo terá informações sobre a localidade inicial do jogador, os blocos do mapas e possíveis inimigos e itens espalhados por ele.

Uma visão detalhada da entrada do arquivo *.map*, assim como a descrição dos atributos utilizados para representar inimigos e itens pode ser encontrada no Apêndice B.2.

2.2.1.3 Visibilidade

É comum no gênero *roguelike* o conceito de visibilidade. Um mapa começa escondido e o jogador só pode ver itens, inimigos e blocos que estejam em seu campo de visão.

Existem porém dois tipos de visibilidade. A visibilidade de terreno, ou visibilidade parcial, identifica o tipo de bloco e itens sobre ele no momento de sua última visualização. Estes blocos costumam ser representados visualmente mais escuros para haver uma diferenciação. A visibilidade total, por sua vez, permite ver tudo que está acontecendo naquele espaço.

A visibilidade é um importante fator de jogabilidade em *roguelikes*, uma vez que muda o modo de agir e pensar dos jogadores. Uma vez que a posição da saída não é conhecida de antemão, eles devem explorar o mapa às cegas, sem conhecimento do que esta por vir, até que encontrem a saída.

O *MetaEngine* pode simular jogos utilizando os dois esquemas de visibilidade. O modo de visibilidade total foi desenvolvido apenas como uma opção de *debug*, utilizando a visibilidade parcial para as simulações.

2.2.1.4 Inimigos e Batalha

A batalha da simulação ocorre através da tentativa de movimento de uma entidade sobre a outra. Por ser uma simulação simples e não um jogo complexo, uma batalha decorre-se apenas pelas Equações 2.1 e 2.2.

$$H = A - D \tag{2.1}$$

onde H é o dano resultante, A o ataque do atacante e D a defesa do defensor.

$$V = V - H \tag{2.2}$$

onde V é a vida do defensor e H o dano resultante.

Eliminando a entidade que chegar a vida zero primeiro. Desta forma, um jogador sempre terá a prioridade de ataque devido ao fato de estar iniciando a ação de movimento. Uma única exceção a esta regra é o caso onde um inimigo mais rápido que o jogador realiza dois ataques em um **turno**.

Os inimigos possuem uma área de observação na qual tomam a iniciativa de avançar e atacar o jogador caso ele adentre esta área. A determinação do caminho a ser percorrido pelo inimigo poderia ser feita de forma direta, devido as áreas de observações dos inimigos serem reduzidas, porém foi utilizado o algoritmo $A*$ dado a sua grande eficiência e rapidez em encontrar o melhor caminho possível a um destino. Esta decisão foi importante no sentido que o algoritmo escolhido também contempla casos de mapas com caminhos estreitos e confusos, caso seja necessário.

2.2.1.5 Algoritmo de menor caminho

O algoritmo A^* (Apêndice C) foi escolhido pela sua grande velocidade de processamento tanto de dados grandes ou pequenos e possuir uma implementação relativamente simples.

Para a implementação no sistema assumiu-se que cada quadrado explorado acarretará em 10 unidades de custo para H e o valor de G é dado por:

$$G = (|X_i - X_f| + |Y_i - Y_f|) \times 10 \tag{2.3}$$

sendo $_i$ e $_f$ indicações das posições para o nó de origem e destino (final).

A Figura 10 demonstra uma iteração do algoritmo. A origem é representada pelo quadrado verde escuro e o fim pelo bloco vermelho. Blocos pretos são obstáculos, enquanto os azuis representam nós explorados e os verdes nós descobertos. F_N é o custo de exploração daquele nó, sendo sempre explorado o nó com o menor custo.

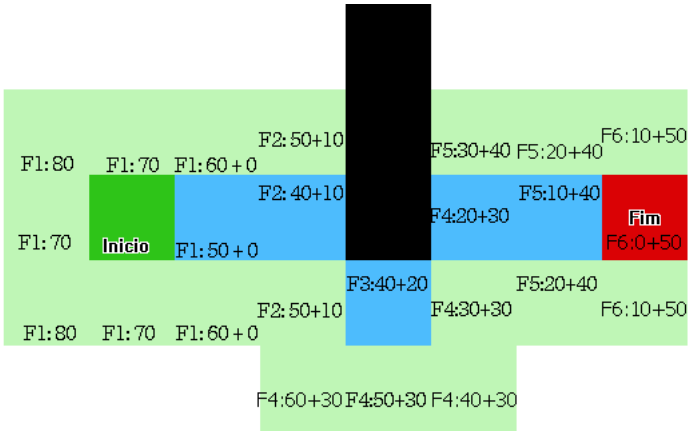

Figura 10 – Algoritmo A*

2.2.1.6 Informações e estados

A simulação possui ainda um sistema de registro (*log*) de mensagens informativas, mostrando danos recebidos e melhorando a jogabilidade da simulação por jogadores humanos. As mensagens serão mostradas no canto superior da tela e escritas sobre o mapa. Estas informações serão apagadas ao toque de uma tecla ou um movimento realizado pelo jogador.

Além das mensagens haverá um painel de estados representando os diversos atributos do jogador como seu Ataque, Defesa, Vida e Dinheiro. Este painel possui fim meramente ilustrativo para que jogadores humanos possam realizar a simulação tendo acesso às informações relevantes para o seu progresso.

2.2.2 Automatização de jogos

Para que a ferramenta *MetaEngine* se torne uma importante ferramenta para auxiliar no teste da qualidade de mapas, inevitavelmente vários jogos deverão ser realizados sobre um mapa. Trazer jogadores humanos para jogar os mapas e testá-los um a um não traria um dos benefícios esperados do trabalho, que é precisamente retirar o usuário do processo inicial de testes e ser capaz de analisar com um certo grau de confiança a qualidade esperada do mapa. Desta forma, as simulações de jogos do sistema foram automatizadas.

2.2.2.1 *Robot* (BOT)

Robot, ou BOT, como é mais comumente chamado, é um termo que costuma ser usado para representar a situação na qual um programa ou algoritmo é utilizado para controlar as ações que normalmente seriam efetuadas por jogadores. BOT's devem ser capazes de receber dados do ambiente em que se encontram e processá-los tomando ações de acordo com alguma regra interna pré-estabelecida. O BOT de fato é uma IA capaz de realizar o controle de ações do jogador.

O sistema utiliza-se de um BOT, adaptado e modificado a partir do código utilizado por *TOME 4* (DARKGOD, 2013), um *roguelike* de código aberto. O BOT é utilizado para simular ações humanas e jogar um mapa inúmeras vezes, com o intuito de recolher uma grande base de métricas em um pequeno intervalo de tempo, se comparado ao tempo que usuários reais levariam.

Uma IA porém, mesmo otimizada, não se compara a um ser humano. Humanos muitas vezes comentem erros ou podem jogar de formas diferentes dependendo de seu humor ou personalidade. Para melhor simular está discrepância entre o processo cognitivo de cada pessoas, o BOT será construído de acordo com parâmetros que o auxiliarão em sua tomada de decisões. Desta forma, com apenas alguns ajustes nos parâmetros, pode-se criar uma IA que simularia um usuário com um perfil mais desafiador, ou um usuário com um perfil mais amedrontado.

2.2.2.2 Perfis

Os principais perfis que podem ser observados em jogadores (obtidos através da análise de diversos vídeos (JEF, 2011) (FRY, 2009) (THEUBERHUNTER, 2012)) podem ser divididos em:

- **Explorador**: Aquele jogador que não quer deixar nada para trás, e gosta de explorar o máximo possível do mapa.
- **Ganâncioso**: Aquele jogador que fará tudo para conseguir mais dinheiro ou itens em um mapa.
- **Corredor**: Aquele jogador que só entra em combate quando extremamente necessário, evitando-os caso possa.
- **Corajoso**: Não deixa nenhum inimigo para trás, enfrentando todos os inimigos em seu caminho.
- **Esperto**: Um meio termo entro os outros perfis: analisa os riscos e evita batalhas as quais está em desvantagem.

- **Apostador**: Assim como o perfil anterior, analisa a situação, porém aceita riscos caso entenda que exista boas recompensas para suas ações.

Cada um destes perfis, apesar de similares, podem afetar jogabilidade de um mapa. Um nível que talvez seja impossível de se completar ao enfrentar todos os inimigos pode ser extremamente fácil para um perfil **Corredor** caso os inimigos do mapa sejam lentos e dispersos.

Apesar de não ser o objetivo do trabalho, a implementação da inteligência artificial do BOT pode ser parametrizada para melhor se aproximar de alguns perfils de jogabilidade identificados e extrair métricas mais próximas a realidade através de parâmetros configuráveis de prioridade em seu código.

2.2.2.3 Algoritmos e parâmetros

Nesta seção serão discutidas as técnicas e algoritmos utilizados para movimentação e tomada de decisões do BOT.

Para melhor simular a randomicidade do processamento humano, o BOT não terá conhecimento qualquer sobre *tiles* que não possam ser visualizados ou já tenham sido visualizados previamente, utilizando os conceitos de visibilidade total e parcial. A inteligência artificial do BOT inicia seu processamento, detectando e obtendo uma lista de objetos de interesse, que serão escolhidos de acordo com o perfil utilizado.

Esta inteligência será chamada pelo *MetaEngine* através do arquivo *playerExplorer.lua*, que está na pasta *data/scripts*, podendo ser alterado pelos usuários da ferramenta, embora esta não seja uma opção recomendável. O *script* será chamado a cada iteração (movimento) do BOT e irá dizer-lher qual a ação deverá ser tomada.

Em seguida é executado um algoritmo similar ao *Depth First Search* (PENTON, 2002). Este algoritmo inicia-se em um ponto e expande sempre o seu nó descoberto menos distante do caminho até que seja encontrado o destino. O algoritmo utilizado segue mesmo processo de exploração do *Depth First Search* de se explorar sempre os nós mais distantes. Porém o ele é utilizado em formas de iterações, sendo que cada iteração do algoritmo é expandido um nível de movimento e criado um mapa de distâncias. Qualquer objeto de interesse ou bloco não descoberto será considerado como um possível destino (Figura 11).

O *script* então procura por vários possíveis destinos. A cada nó descoberto é feita uma análise para identificar que tipo de bloco e que objetos estão sobre ele, a partir das seguintes regras:

- **Bloco já visto e sem itens ou inimigos** - Nada a fazer.
- **Bloco não visto** - Adiciona a lista de *tiles* de interesse.

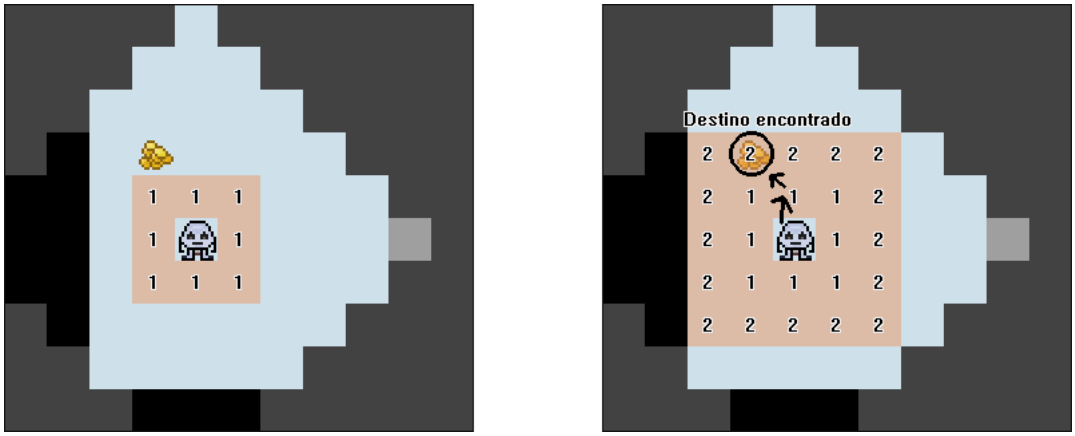

Figura 11 – *Depth First Search* em iterações por distância

- **Bloco com itens** - Adiciona a lista de *itens* avistados.
- **Bloco com inimigos** - Adiciona a lista de inimigos avistados.
- **Bloco de saída do mapa** - Adiciona a lista de alvos prioritários.

Vale a pena lembrar que, uma vez que o bloco não é visível, o BOT não terá conhecimento sobre o que há nele, podendo até mesmo ser uma parede.

Este processo continua seqüencialmente junto ao descobrimento de novos blocos. A cada novo nó aberto, o BOT irá realizar uma verificação para confirmar se ele chegou ao seu objetivo ou não.

Para garantir que prioridades de escolha possam ser estabelecidas mais tarde é necessário que sejam avistados mais do que um único destino durante a descoberta, como é o caso dos algoritmos de descoberta de caminho. Para se obter tais informações, o algoritmo, ao avistar o primeiro objeto de interesse, irá marcar-se como concluído e executará um número N de iterações adicionais, guardando os novos destinos em uma lista para garantir que existam diversos possíveis alvos e que o BOT possa realizar uma análise mais abrangente.

Após rodar esta etapa do algoritmo, o *script* possuirá suas devidas listas de *tiles* avistados, itens e inimigos, e iniciará a etapa de processamento destes dados para escolher o seu novo destino (Figura 12).

O BOT divide a prioridade de suas escolhas de acordo com a proximidade, no tipo e nos *parâmetros de ganância* especificados pelo tipo de perfil. A IA também tem o conceito de blocos solitários, *singlets*, e tenta priorizar para que não sejam deixados para trás quando não muito custosos (evitando assim movimentos adicionais). Blocos solitários se identificam por um nó do mapa não visto cercado apenas por blocos visíveis. Muitas das vezes este pedaço do mapa será uma área passável, podendo haver itens. Esta situação tende a gerar um custo muito alto de movimento para voltar e re-explorar um

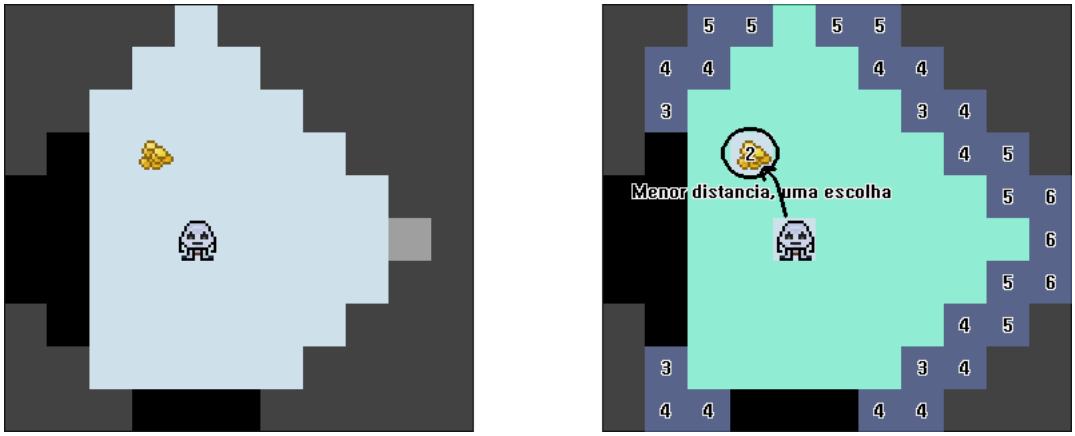

Figura 12 – Passos simplificados demonstrando o encontro dos blocos pela IA

bloco esquecido caso seja um perfil explorador ou, num caso extremo, tal bloco pode ser o bloco de saída do mapa.

A ordem de prioridade de escolha de alvos, sem a alteração de quaisquer *parâmetros de ganância*, são:

1. *Singlets* - Blocos solitários
2. Itens
3. Inimigos
4. Blocos inexplorados

O primeiro objeto que entrar na lista de escolhas terá o valor de escolha mínimo igual a sua distância, isto é, todos as outras listas analisadas serão somente adicionadas a lista de possíveis escolhas caso estejam a uma distância mínima menor que o primeiro objeto prioritário encontrado. Porém, para a criação dos diversos perfis é adicionado um *parâmetro de ganância* para adicionar objetos que estão mais distantes do que a distância d estipulada, se o objeto tem grande importância para o perfil desejado. A distância a ser considerada para o algoritmo de inserção será então computada pela Equação 2.4:

$$D = d - g \tag{2.4}$$

sendo D sua nova distância, d a distância real do objeto e g o parâmetro de ganância do tipo de objeto em questão.

Desta forma é possível alterar a forma com que a IA irá colocar as suas escolhas de acordo com o resultado esperado daquele perfil. Por exemplo, um perfil explorador e ganancioso terá altos parâmetros de ganância para itens e possuirá um maior número de iterações extras para melhor identificar itens nas redondezas, enquanto um perfil Corajoso possuirá altos parâmetros com inimigos (Figura 13).

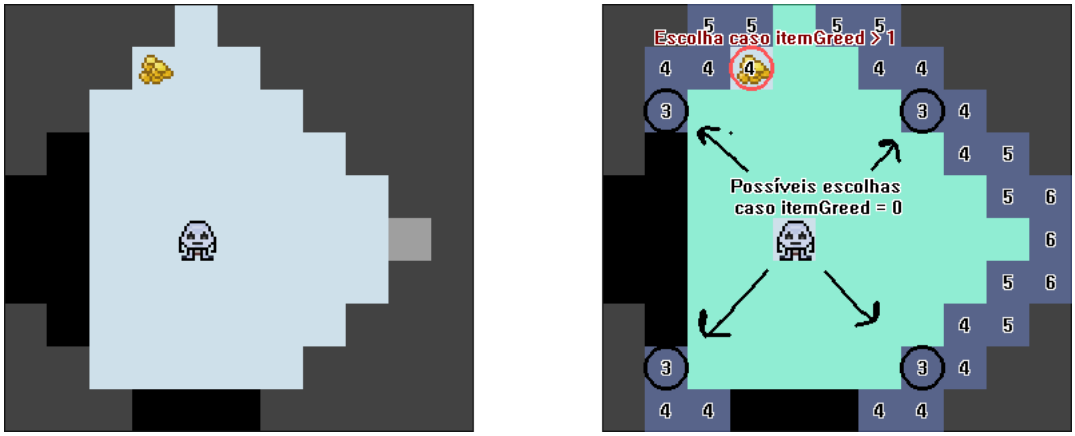

Figura 13 – BOT - Alternativas e parâmetros de ganância

Ao final do processo, caso não hajam novos meios de filtrar a lista de escolhas restantes, é então escolhido um alvo aleatório dentro das possíveis escolhas.

Por fim, o *script* chamará a função da classe do jogador de dentro do sistema e pedirá que seja construído o caminho até o alvo encontrado.

Na próxima ação que o BOT for tentar executar, será antes checado se é necessário que haja uma nova análise dos dados para escolha de um novo alvo ou se a IA deverá prosseguir até o seu alvo antes de iniciar o processamento novamente. Isto é testado pela verificação da existência de uma rota em progresso: caso não haja, simplesmente chama-se a análise novamente. Porém, caso seja encontrada uma rota, o *script* ainda fará verificações para otimizar a inteligência e garantir que não esteja andando para um grupo de inimigos ou beco sem saída, por exemplo.

Desta forma, mesmo havendo rotas presentes para o jogador, caso aviste algum **novo** inimigo em seu caminho ou realize um movimento de tal forma que o bloco destino esteja visível porém não note a presença de quaisquer novos blocos, irá ser realizada uma nova análise e a escolha de novos destinos (Figura 14).

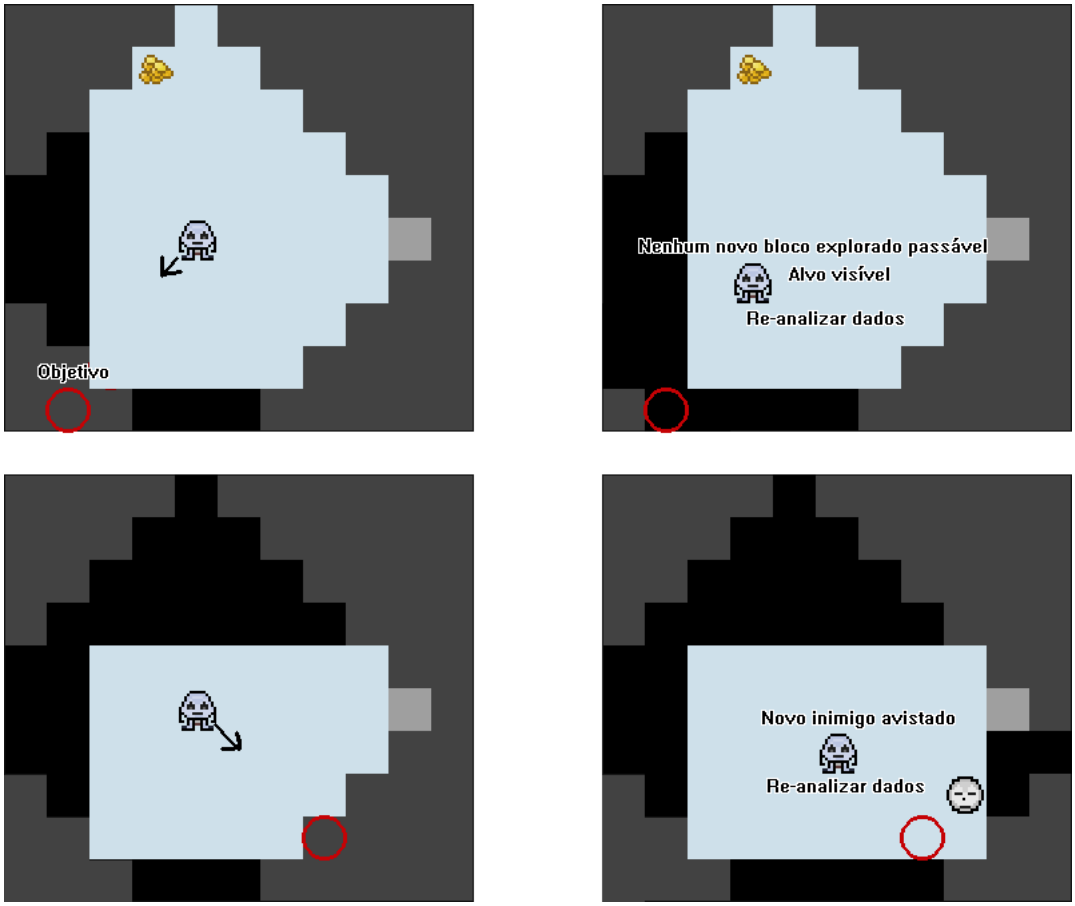

Figura 14 – Situações de nova análise dos dados pela IA

2.2.3 Métricas

A ferramenta *MetaEngine* possui a capacidade de extrair uma série de métricas ao decorrer de um jogo simulado. As métricas são recolhidas e armazenadas, separando-as para cada mapa e rodada (por rodada entende-se uma partida simulada) em que foram testadas.

Desta forma, dadas as métricas A, B, C e a realização de três jogos em um dado, mapa a ferramenta é capaz de salvar informações sobre os valores das métricas A, B, C para cada uma das rodadas e estimar também a qualidade total através dos valores obtidos e as esperanças dos valores ideais para o mapa.

O primeiro passo a ser feito para a extração das métricas e estimativa da qualidade é inserir no *MetaEngine* os valores esperados ideais para o mapa de acordo com a expectativa do usuário da ferramenta (possivelmente o *game designer* da equipe). Não existe um valor unificado para a qualidade um mapa: um mapa pode ter diferentes objetivos, ficando a cargo do desenvolvedor decidir que tipo de mapa ele espera. Se é um mapa para

um jogo rápido e casual, provavelmente será um mapa com um pequeno tempo de duração e dificuldade. Se for um mapa para um jogo com mais ação, provavelmente se espera que haja uma maior quantidade de inimigos e espaço livre para a movimentação.

2.2.3.1 Métricas utilizadas

As principais métricas identificas foram:

1. **Movimentos**:
   Indica a quantidade de passos realizados. Através desta métrica é capaz de obter-se a duração de um jogo. O tempo utilizado para a realização de um turno em um jogo do gênero possui uma grande variação: dependendo do momento em que o jogador se encontra, ele pode demorar vários segundos em um único turno, em situações de batalha, ou completar vários turnos em 1 segundo, para corredores e áreas já vistas. A conversão de turnos-segundos utilizada foi 1:1, isto é, 1 turno (movimento) equivale a 1 segundo. Futuros testes com usuários reais serão necessários para ajustar esta razão de conversão.

2. **Itens coletados**:
   Esta métrica indica a quantidade de itens coletados, podem caracterizar se o nível criado é recompensador o suficiente para o jogador. Esta métrica pode simplesmente registrar a quantidade de itens coletados, ou utilizar um fator peso para representar a importância do item coletado para o jogador. Pode também dividir-se em métrica sobre a quantidade de itens de atributos, que aumentam as habilidades do jogador, ou métrica sobre a quantidade de itens de recuperação.

3. **Dinheiro coletado**:
   Indica a quantidade de dinheiro coletado ao longo do jogo. Permite a visualização de quão recompensador foi o jogo, e serve para checar se os mapas tem um progresso consistente na evolução do jogo. Por exemplo, para manter um jogador motivado em um jogo é comum que se tenha um progresso constante: grandes variações neste fluxo pode obter efeitos negativos, como a desmotivação do jogador, ou deixar o jogo muito fácil, habilitando a compra de melhores itens de forma precoce.

4. **Inimigos:**
   Esta métrica indica a quantidade de inimigos derrotados. Também pode ser expandida para a quantidade inimigos vistos, porcentagem de inimigos vistos e/ou derrotados. Pode ser utilizada para o balanceamento dos mapas, garantindo que os mapas não possuam uma grande dificuldade.

5. **Vitórias:**
   Quantidade de vitórias e derrotas que um BOT obteve em N simulações de parti-

das em um mapa. Estabelece o quão aceitável o mapa pode ser e a progressão de dificuldade. Jogos casuais, por exemplo, possuem pequeno aumento de dificuldade entre mapas e normalmente costumam ter vitórias fáceis, enquanto jogos *hardcore* costumam ser extremamente difíceis. O jogo *Dungeon Crawl Stone Soup*[5], voltado ao *roguelike* clássico por exemplo, realizou uma competição (ELLIPTIC, 2013) com 1.749 participantes, com uma média de vitórias de 1.37% e uma média de duração de jogo de 14.8 horas.

Com o avanço do trabalho, dividiu-se as métricas em métricas primitivas e compostas.

1. **Primitivas:**
   São métricas que podem ser obtidas de forma direta e não dependem de quaisquer outras métricas para gerar seus dados.

2. **Compostas:**
   São métricas que são formadas através da composição de uma ou mais métricas simples.

Dados estas definições, foram elencadas diversas métricas que podem se tornar relevantes a um desenvolvedor de mapas. Algumas das métricas primitivas foram coletadas somente para suprir métricas compostas e não necessáriamente produzem valores relevantes para a aferição da qualidade. A utilização ou não utilização das métricas fica a cargo do utilizador da ferramenta.

Abaixo segue a relação das métricas que são extraídas durante a simulação:

Primitivas

a) Passos dados

b) *Tiles* vistos

c) *Tiles* totais

d) Itens vistos

e) Itens usados

f) Itens de recuperação totais

g) Itens de recuperação usados

h) Itens de *status* totais

i) Itens de *status* usados

[5] http://crawl.develz.org/wordpress

j) Inimigos vistos

k) Inimigos derrotados

l) Inimigos totais

m) Vida recuperada

n) Dano causado

o) Dano recebido

p) Ataques executados

q) Ataques recebidos

r) Dinheiro obtido

s) Dinheiro total

Compostas

a) Porcentagem de tiles vistos

b) Porcentagem de inimigos vistos

c) Porcentagem de inimigos derrotados

d) Porcentagem de itens obtidos

2.2.3.2 Análise de métricas

Uma parte significativa deste trabalho foi estabelecer uma forma quantitativa de qualificar uma série de dados obtidos dos mapas como valores bons ou ruins, de acordo com um intervalo e através da compilação destes valores estimar se um dado atinge a qualidade esperada de acordo com os valores estabelecidos esperados.

A primeira proposta para a resolução deste problema foi a utilização das hipóteses de teste de comparação através do teste t de Student (Apêndice D). Este teste pode ser usado para verificar se duas amostras possuem médias ou variâncias estatisticamente iguais. Porém, os valores obtidos com este testes se mostraram muito tendenciosos e irregulares em relação à proposta deste trabalho.

Outra abordagem avaliada foi o teste Chi Quadrado (Apêndice E) o qual, dado dois conjuntos de amostras, indica a chance de serem iguais. Apesar de parecer ideal, este teste necessita de valores concorrentes. Isto significa que, para o teste Chi Quadrado funcionar efetivamente, deveria-se ter uma amostra de valores ideais para se comparar com os valores obtidos. Foi experimentado gerar uma série de valores ideais através de uma distribuição normal para tentar obter-se a amostra ideal, porém, para este teste a quantidade de valores gerados alterava a saída, uma vez que este método compara valores pela freqüência absoluta dos valores dos conjuntos.

Uma vez que os testes de comparações mais conhecidos para se comparar amostras não se adequaram as necessidades do trabalho, foram buscadas formas alternativas para medir os valores das métricas.

Uma das formas alternativas buscadas para se resolver este problema foi a utilização de distribuições normais de probabilidade e qualificar a porcentagem de chance dos resultados obtidos pertencerem a ela. Esta forma de avaliação porém apresenta em sua definição um problema de adequação.

Uma distribuição normal de probabilidade pressupõe-se que os valores estejam normalmente distribuídos, isto é, quanto mais próximo da média populacional, maior será a concentração de resultados. Isto não é necessariamente verdadeiro em jogos, pois a alternativa de explorar primeiro um caminho A ou um caminho B pode mudar significativamente o resultado do jogo. Por este motivo, a utilização da porcentagem do desvio normal padrão puramente caracterizou uma forma adequada de avaliar as métricas coletadas.

Para melhor expandir os horizontes de possibilidades foram então testadas três novas abordagens para o tratamento das métricas. A primeira foi utilizar um intervalo de aceitação pré-definido, isto é, qualquer valor medido que estiver neste intervalo acarretaria no acréscimo de 1 ponto na pontuação final, que por sua vez seria dividida pelo número de valores amostrados.

Enquanto esta abordagem mostrou-se aceitável, os seus valores são rígidos de tal forma que se um mapa possuir muitos valores concentrados em uma área próxima, mas fora do intervalo de aceitação uniforme, ele será classificado com um mapa ruim, o que não é necessariamente verdade.

A segunda abordagem levou em consideração que a qualidade de valores dentro de um intervalo de aceitação estabelecido varia de acordo com a sua proximidade do centro. Utiliza-se de um intervalo fixo assim como a anterior, mas, a quantidade de pontos acrescidos à pontuação final de cada um acerto varia de acordo com a sua proximidade em relação a média do intervalo. Assim como o primeiro método, este atribuiu pontuação a valores fora do intervalo e por isto permanece os mesmo problemas, apesar de obter-se uma nota melhor para valores centralizados, esta nota nunca é maior que obtida com a abordagem anterior, tornando-a inviável e com valores, em média, muito baixos de qualidade.

A terceira abordagem, foi a que demonstrou valores mais próximos das qualidades esperadas de uma mapa e a escolhida para o *MetaEngine*. Ela faz uso de uma das mais importantes características de uma distribuição normal: a probabilidade dos valores de uma dada distribuição estarem em até um dado desvio padrão de sua média.

Esta abordagem testa a probabilidade dos valores estarem centrados dentro do desvio padrão, atribuindo uma pontuação de acordo com a proximidade. Sabe-se que

aproximadamente 99,9 por cento dos valores estão em até três desvios padrões da média e como a série normal é simétrica, aproximadamente 49,9 por cento dos valores para cada um dos lados a partir da média.

O usuário deve alimentar o *MetaEngine* com os valores mínimos e máximos aceitáveis para cada medida, e estes valores são utilizados para computar as médias e desvios padrões ideais, utilizando as Equações 2.5 e 2.6. Considerando os valores esperados máximo e mínimos $Dmax_i$ e $Dmin_i$ para uma dada métrica i, pode-se obter os valores para a média μ_i e o desvio padrão σ_i ideais.

$$\mu_i = \frac{(Dmax_i + Dmin_i)}{2} \tag{2.5}$$

$$\sigma_i = \frac{(Dmax_i - Dmin_i)}{2} \tag{2.6}$$

Desta forma todos os valores que estiverem dentro do intervalo estipulado pelo usuário estarão a no máximo 1 desvio padrão da média da distribuição normal ideal.

Com estes valores é construída uma função de densidade de probabilidade (Figura 15) dos valores ideais utilizando o método da distribuição normal (MONTGOMERY; RUNGER, 2003).

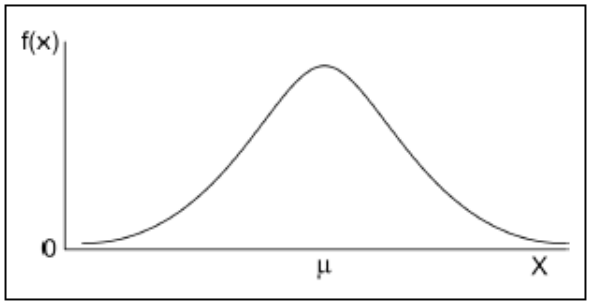

Figura 15 – Distribuição normal de probabilidade

De posse da média e do desvio padrão, pode-se obter o valor do escore Z para uma medida x obtida através da Equação 2.7.

$$Z = \frac{x - \mu}{\sigma} \tag{2.7}$$

Com o valor de probabilidade oferecido por Z, é possível calcular a probabilidade do valor medido estar contido em um dado intervalo. A qualidade aferida por este método se baseia em obter a chance de um extremo da função de probabilidade de densidade até o valor obtido, e calcular a porcentagem que se aproximou de chegar até a média.

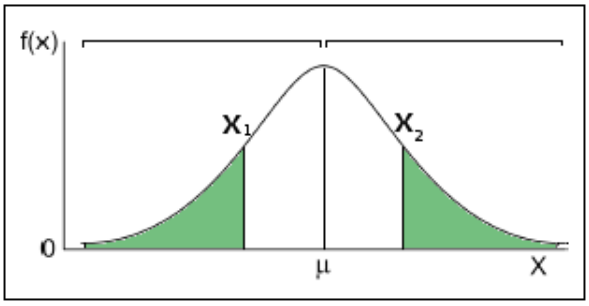

Figura 16 – Áreas desejadas

Portanto, caso o valor obtido seja menor que a média, o valor da densidade desejado será o representado pela área verde do X_1 na Figura 16 e se maior que a média, a área a direita. A qualidade por fim é representada pela cobertura da área obtida até o centro do gráfico, isto é, a média.

$$Q = \frac{P(X \leq x)}{0,5} \tag{2.8}$$

se X menor que a média e

$$Q = \frac{P(X > x)}{0,5} = \frac{(1 - P(X \leq x))}{0,5} \tag{2.9}$$

se X maior ou igual que a média.

O valor de Q representa a qualidade em uma escala de zero a um, onde o valor um representa 100% de qualidade. Esta forma de medição de qualidade garante que todos os valores no intervalo especificado pelo usuário estarão, no máximo, a um desvio padrão da média.

A qualidade total do mapa será a média ponderada dos valores de qualidade de cada uma das n métricas individuais pelo valor de peso p_i estabelecido para cada uma delas, conforme a Equação 2.10.

$$Q_{total} = \frac{\sum Q_i \cdot p_i}{\sum p_i} \tag{2.10}$$

2.3 Ferramenta de criação de mapas - *MapBuilder*

A ferramenta *MapBuilder* foi desenvolvida no *framework* Qt utilizando a linguagem C++ para ajudar no desenvolvimento e observação de mapas, uma vez que construí-los através das regras definidas pelos arquivos *.map* se torna extremamente ineficaz e cansativo em um processo manual. A ferramenta permite a criação e alteração de mapas, sejam eles feitos a mão ou mapas gerados proceduralmente em tempo de execução. A Figura 17 mostra a construção de mapas pelo editor.

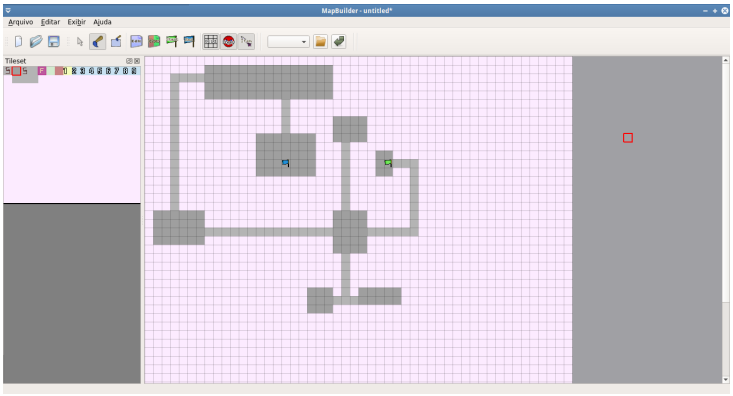

Figura 17 – Ferramenta auxiliar de construção de mapas

A interface é adaptável e permite a utilização de janelas flutuantes ou janelas fixadas nas bordas do programa, dando o usuário a liberdade de criar a sua área de trabalho do jeito que lhe seja mais aprazível. A interface de edição possui 3 modos de operação para a alteração do mapa que servem para melhor orientar o usuário da ferramenta em relação ao que está sendo feito.

O primeiro é o modo de edição de gráficos. Neste modo, o usuário tem acesso a utilitários como a ferramenta "pincel" para escrita dos blocos no mapa unitariamente e a ferramenta "retângulo" para a criação de blocos em área. Isto permite a criação mais efetiva do mapa, dando também ao usuário a opção de e selecionar os blocos que desejam ser pintados um a um, ou em grupos.

O segundo modo é, apresentado pela Figura 18, diz respeito à edição de entidades. Neste modo o usuário poderá criar e alterar entidades para o seu mapa, criar entidades padrões, copiá-las e posicioná-las da forma que desejar. As entidades estão divididas em 3 grandes tipos genéricos: Itens, Inimigos e Dinheiro.

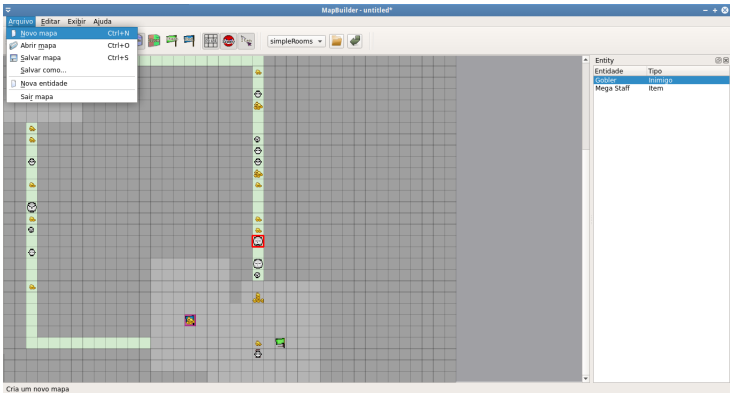

Figura 18 – Modo de edição Entidades e menu principal

O último modo é o modo de caminhos, mostrado pela Figura 19. Com ele é possível visualizar quais blocos são passáveis e quais blocos não serão. Ferramentas para edição rápida como o "retangulo"e a cópia em grupo também se aplicam para esse modo.

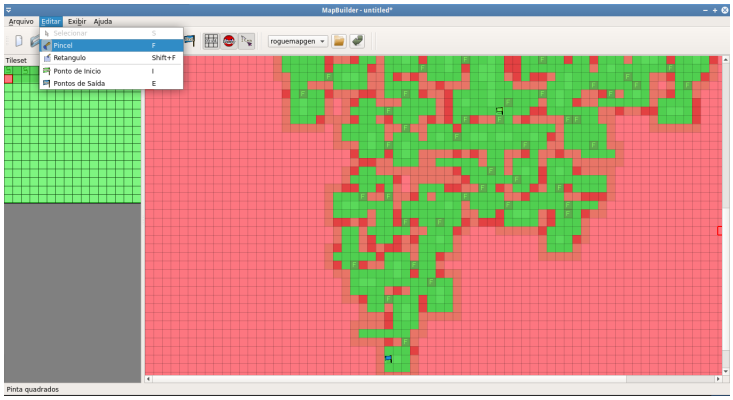

Figura 19 – Modo de edição de caminhos e ferramentas

Todos os utilitários da aplicação podem ser acessados através de atalhos de teclado, e existem algumas opções para melhor visualização, como a opção de mostrar ou não a malha de quadrados. Por fim, existem dois botões sinalizando bandeiras que indicam aonde serão as entradas e saídas do mapa, isto é, aonde o jogador começará o jogo e aonde poderá chegar para terminá-lo.

Como um outro extra para a ferramenta, foi adotado uma biblioteca *qscintilla2*, que fornece uma API para a representação de algumas funcionalidades de editores de texto e de código. Com ela foi possível construir uma mini-IDE embutida na ferramenta com a capacidade de interpretar comandos e sintaxes da linguagem *Lua* para a criação de *scripts* procedurais e visualizar os resultados simultaneamente, indicando também os erros que ocorreram durante a interpretação dele, mostrado pela Figura 20.

Para testar a utilização da ferramenta e também para obter-se uma variação de mapas procedurais para o trabalho, foram escritos alguns *scripts* para geração de mapas, baseados em algoritmos descritos em artigos[6] de geração producedural e de adaptações de algoritmos utilizados em jogos e simulações do conheçido criador de *Cookie Clicker*, Orteil[7].

Os códigos procedurais construídos feitos na linguagem *Lua* e são adaptações dos métodos das fontes descritas acima. Tais *scripts* se encontram no Apêndice F.

[6] Geração de mapas procedurais com árvores BSP - http://gamedevelopment.tutsplus.com/tutorials/how-to-use-bsp-trees-to-generate-game-maps–gamedev-12268

[7] Simulador de mapas procedurais feito em javascript - http://orteil.dashnet.org/experiments/dungeongenerator/

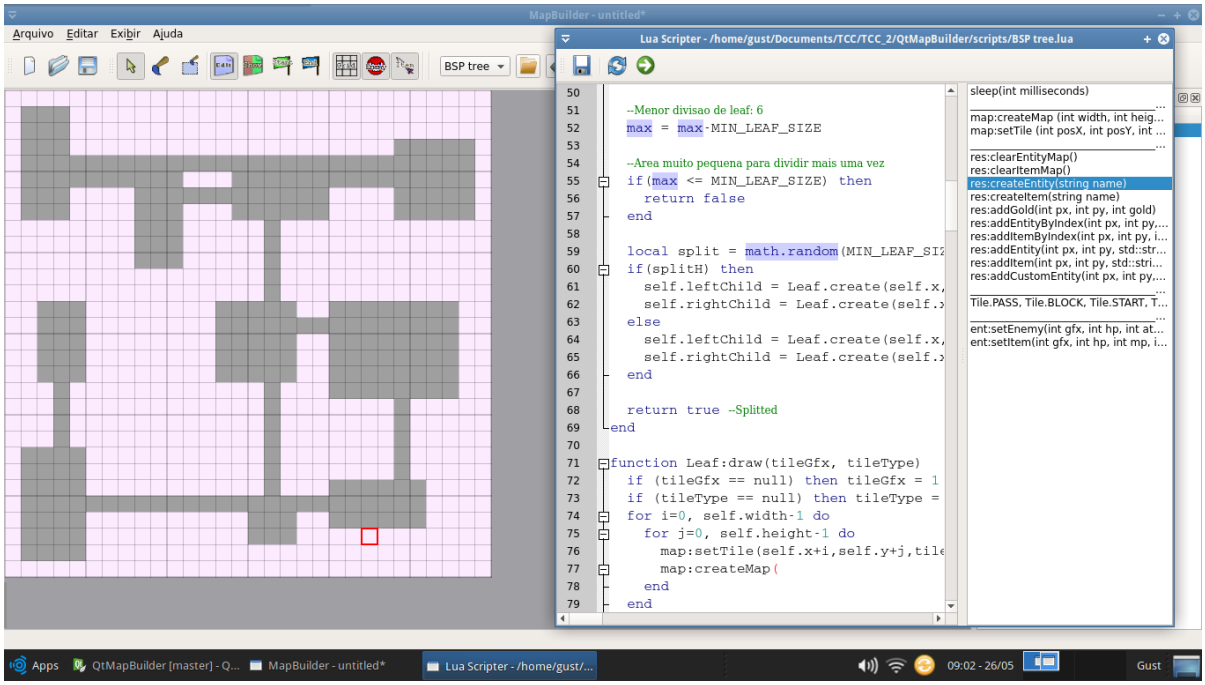

Figura 20 – Edição de códigos dentro da ferramenta auxiliar, botões acima da janela executam o código em tempo real

3 Resultados

Esta seção será dedicada a avaliação dos resultados do trabalho, das avaliações obtidas através de testes de usuários e das simulações automatizadas pela ferramenta.

3.1 Técnicas de avaliação de métricas

Como evidenciado no desenvolvimento, foram testadas diversas formas de avaliação de métricas, algumas com mais sucesso do que outras. Uma das hipóteses para essa dificuldade de avaliação foi o fato de que os valores coletados não serem, em geral, normalmente distribuídos. Outro ponto questionado durante a avaliação foi sobre o quão prejudicialmente uma métrica coletada de um jogo incompleto (derrota do jogador) afetaria a métrica de qualidade. Por exemplo, evitar um inimigo forte e tomar uma rota alternativa, ou tentar derrotá-lo e morrer pode gerar grandes divergências na quantidades de passos movidos.

Foram coletados os dados de todas as métricas de 400 rodadas de um jogo através da inteligência artificial utilizada. Os dados sobre ataques recebidos, dano causado e passos realizados podem ser visualizados pela Figura 21, Figura 22 e Figura 23 respectivamente. O eixo X dos gráficos a seguir representam os valores medidos e o eixo Y representa a quantidade de vezes que as métricas foram medidas para aqueles valores. Os gráficos mostrados mostram as métricas coletadas somente considerando jogos onde o jogador obteve a vitória a direita e considerando todos os jogos a esquerda.

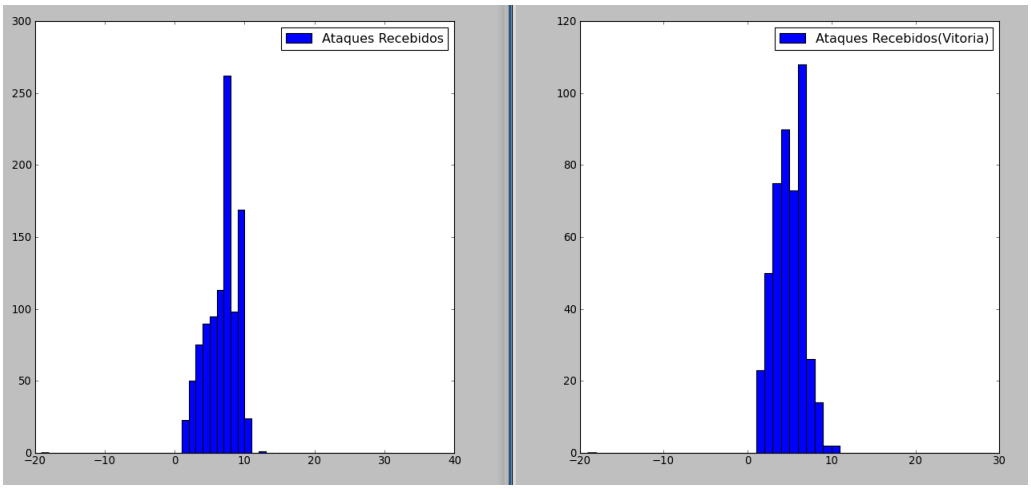

Figura 21 – Comparação de métricas: Ataques recebidos

Como é possível observar, os valores de medidas considerando somente vitórias e considerando todos os valores não altera de forma muito significativa o resultado final. A

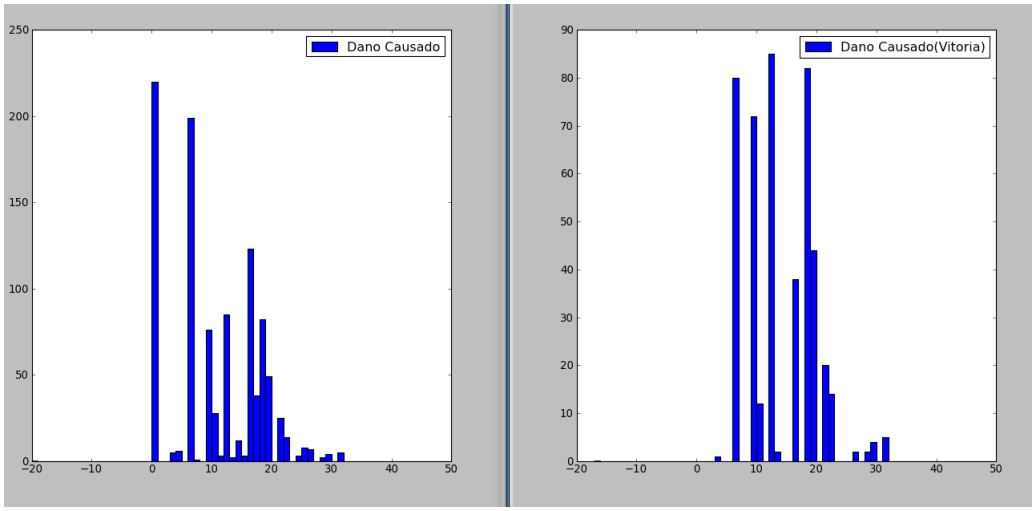

Figura 22 – Comparação de métricas: Dano causado (mapa 1)

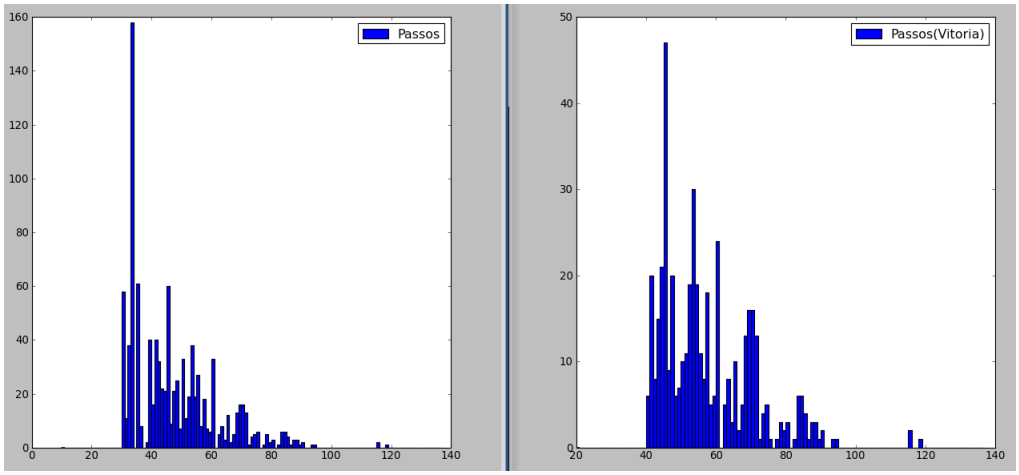

Figura 23 – Comparação de métricas: Passos realizados (mapa 1)

freqüência de valores aumenta devido a considerar mais resultados, e há algumas variações, porém, a forma e concentração do gráfico permanecem consideravelmente semelhantes.

Uma das situações hipotetizadas para um mapa é que seus dados coletados não convergem a um único ponto e sim a diversos picos de dados clusterizados (agrupados). Por exemplo, dado que um jogador tome uma rota que não possua muitas bifurcações, e possível inferir que a quantidade de passos que ele realizará para atingir seu objetivo tenda a se aproximar de um valor enquanto um outro mapa com duas bifurcações, uma delas sendo um longo beco sem saída, faça os valores a se aproximar de picos, os valores dos quais o jogador tomou o caminho certo e os dos quais o jogador tomou o caminho errado e teve que retornar aumentando consideravelmente a sua quantidade de passos. Para verificação desta hipótese, foram extraídas medidas sobre outros dois mapas utilizando desta vez 1.000 medidas. Os valores de passos realizados de ambos os mapas podem ser observados pela Figura 24 e a Figura 25.

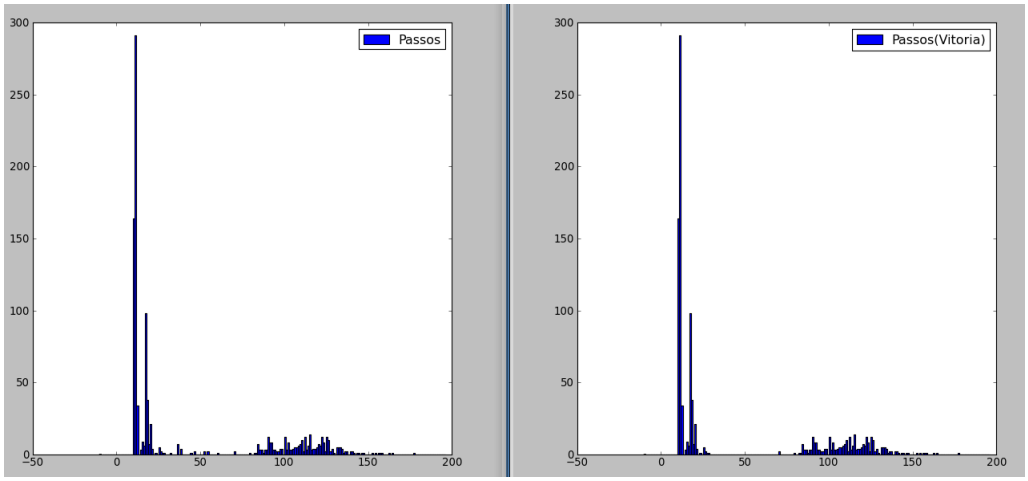

Figura 24 – Comparação de métricas: Passos realizados (mapa 2)

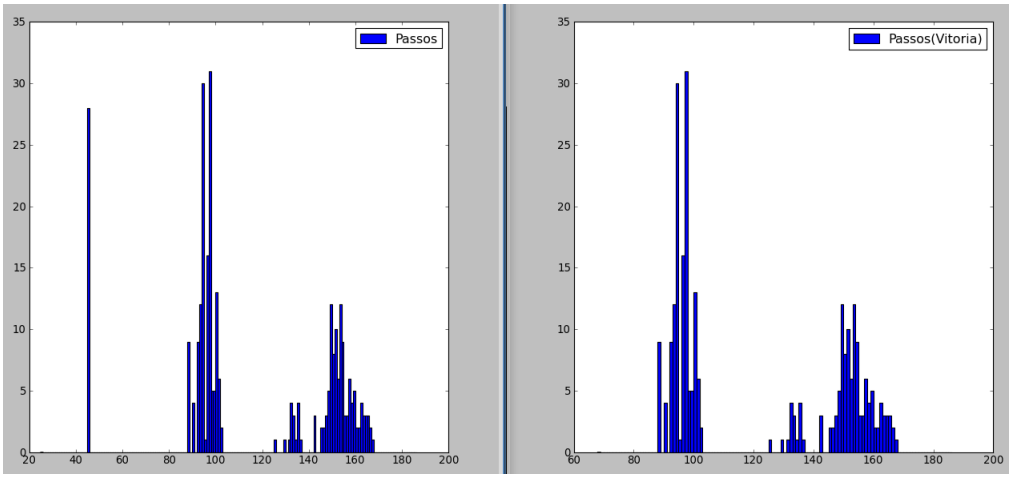

Figura 25 – Comparação de métricas: Passos realizados (mapa 3)

Pode-se observar que há de fato uma clusterização de valores e dois pontos em ambos estes mapas. Isto pode afetar a medida de qualidade do mapa caso não sejam tomados as devidas precauções. A abordagem para o cálculo da métrica descrita no desenvolvimento tenta levar em consideração possíveis erros como este de forma que, apesar de resultarem em um declive de qualidade, esta redução não será tão brusca como outros métodos experimentados, devido a consideração da qualidade estar na distância do desvio padrão do ideal.

Por fim, através deste último gráfico abaixo (Figura 26) que foi tirado da mesma medida que o gráfico (Figura 25), pode-se notar que mesmo com a clusterização dos Passos, o mesmo mapa não mostrou a clusterização de valores de ataques executados.

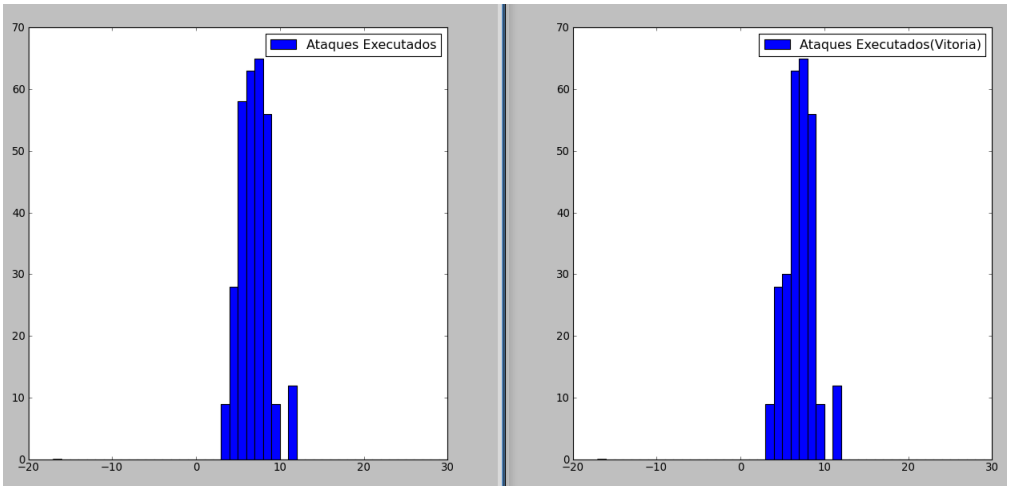

Figura 26 – Comparação de métricas: Ataques executados (mapa 3)

3.2 Avaliação de mapas

Para a verificação e análise da qualidade de diferentes mapas criados proceduralmente foram projetados dois simples algoritmos, um deles baseado e adaptado para Lua de (HELY, 2013). Apesar da simplicidade dos algoritmos, ambos suportam a parametrização de valores para sua criação, o que ajudou a testar a influência da geração parametrizada sobre diferentes perspectivas.

A avaliação a seguir visou observar a mudança de qualidade de mapas gerados para a mesma meta sobre diferentes parâmetros e diferentes algoritmos procedurais. Será possível também observar as peculiaridades de cada algoritmo através da grande massa de dados que foi gerada.

O primeiro algoritmo é muito simples e apenas constrói salas quadradas interligadas, com a chance de criação de inimigos e itens em seus corredores. Este algoritmo não possui um tratamento real de balanceamento e as salas por vezes são criadas dentro uma das outras. O *script* de geração deste algoritmo pode ser encontrado no Apêndice F.1. Possui os seguinte parâmetros: quantidade de salas, tamanho máximo e mínimo das salas

O segundo algoritmo utiliza árvores de particionamento binário de espaço para a criação de salas ideais, criando uma boa e uniforme disposição espacial do mapa. Os inimigos e itens deste algoritmos também não possuem um tratamento real e são apenas gerados aleatóriamente dentro das salas. O *script* de geração deste algoritmo pode ser encontrado no Apêndice F.2. Possui os seguintes parâmetro: tamanho mínimo e máximo de salas (folhas da árvore)

Os valores considerados como ideais foram os seguintes:

- Passos: 80-200, peso: 1,0

- *Tiles* vistos: 200-450, peso: 1,0
- Inimigos Derrotados: 2-6, peso: 0,2
- Itens usados: 4-15, peso: 0,1

Os outros parâmetros de métricas foram desprezados e a importância das métricas utilizadas em relação a inimigos e itens foi reduzida devido a grande variância dos mesmos presente nos algoritmos. Ou seja, está sendo avaliado aqui fortemente o deslocamento e disposição espacial do mapa.

Para o primeiro teste, denominado Experimento 1, foram realizados testes em 10 mapas de cada algoritmo procedural utilizando 50 rodadas simuladas sobre as seguintes parametrizações (Tabela 2). Os seguintes resultados foram gerados (Tabela 3).

Algoritmo	**Sala mínima**	**Sala máxima**	**Quantidade de salas**
Algoritmo 1	3x3	6x6	8 salas
Algoritmo 2	3x3	8x8	não possui o parâmetro

Tabela 2 – Parâmetros utilizados - Primeiro experimento

Jogos	**Qualidade Algoritmo 1**	**Qualidade Algoritmo 2**
1	0,24	0,53
2	0,61	0,50
3	0,41	0,28
4	0,28	0,35
5	0,26	0,32
6	0,04	0,15
7	0,42	0,62
8	0,47	0,46
9	0,33	0,49
10	0,13	0,32

Tabela 3 – Qualidade entre algoritmos

Foi observado que todos os valores do Algortimo 2 que obtiveram nota inferior a 0,3 tiveram um indice de mortes do jogador de mais de 50 por cento, o que reforça o fato de que jogos onde o jogador não obteve a vitória influenciam muito a classificação destes mapas. Isto está relacionado com a forma com que o algoritmo monta seus mapas, deixando a saída sempre no ponto mais distante da entrada, com poucas rotas alternativas. Contrariamente, houveram mapas no Algortimo 1 que obtiveram notas baixas apesar de terem 100 por cento de vitórias, o que pode ter ocorrido pela falta de verificação espacial

do Algortimo 1 que permite, em casos raros, que a saída seja colocada ao lado ou muito próximo à entrada.

Alguns mapas gerados criaram impossibilidades de vitória devido ao contraste de força dos inimigos, gerados resultando numa qualidade Q de aproximadamente zero.

O próximo teste, denominado Experimento 2, foi realizado utilizando os parâmetros da Tabela 4.

Algoritmo	**Sala mínima**	**Sala máxima**	**Quantidade de salas**
Algoritmo 1	6x6	10x10	5 salas
Algoritmo 2	5x5	12x12	não possui o parâmetro

Tabela 4 – Parâmetros utilizados - Experimento 2

Jogos	**Qualidade Algoritmo 1**	**Qualidade Algoritmo 2**
1	0,14	0,29
2	0,12	0,20
3	0,24	0,50
4	0,27	0,17
5	0,48	0,18
6	0,47	0,23
7	0,53	0,37
8	0,41	0,08
9	0,33	0,19
10	0,51	0,48

Tabela 5 – Qualidade entre algoritmos 2 - Salas maiores

A Tabela 5, que apresenta os resultados do experimento, mostra que houve uma visível queda de qualidade no segundo algoritmo e a mesma característica se preservou, ou seja, todos os mapas com a qualidade inferior a 0,2 tiverem mortes do jogador excedendo 50 por cento. O primeiro algoritmo porém sofreu um aumento na qualidade, ou seja, como os dois fatores governantes alocados para a métrica ideal foram passos e blocos vistos, a característica do algoritmo 1 de gerar grandes salas sobrepostas. Apesar de obter um menor número de passos em geral, o jogador passa a observar mais blocos por movimentos (somente são contados blocos visíveis e passáveis).

Apesar dos resultados da qualidade geral obtidos através destes testes não terem sido elevados, a análise dos resultados gerados apontam diversas características proveniente de cada um dos algoritmos de forma rápida e sem a intervenção direta de um grupo de usuários de teste.

3.3 Avaliação das métricas colhidas de usuários

A seguir foi realizado um teste com a participação de 5 usuários humanos. Para o teste realizado foram escolhidos 5 mapas dentre os quais, 4 foram gerados aleatóriamente pelo mesmo Algoritmo 1 apresentado na seção anterior, e somente um dos mapas foi criado manualmente seguindo as regras de criação do *script*, isto é, utilizando somente salas quadradas para que não fique tão evidente a diferença de geração entre eles. Numa situação real de desenvolvimento, os algoritmos procedurais a serem utilizados serão bem tratados e confeccionados a fim de que os mapas sejam mais irregulares, mas como o objetivo deste trabalho não é a criação de *scripts* procedurais, o algoritmo utilizado para testes utiliza apenas salas quadradas interligadas.

O primeiro mapa testado permitiu obsevar que os usuários receberam pontuações de qualidade similares. A Tabela 6 apresenta os resultados de algumas das métricas coletadas nas partidas jogadas pelos 5 usuários.

Métrica	**Usuário 1**	**Usuário 2**	**Usuário 3**	**Usuário 4**	**Usuário 5**
Passos	41	97	89	118	167
Tiles vistos	320	458	443	464	547
Inimigos derrotados	0	1	1	1	2
Itens coletados	4	17	15	15	19

Tabela 6 – Métricas obtidas

Considerando somente um jogo e os mesmos parâmetros de métricas ideais de entrada do Experimento 1 e 2, os usuários obtiveram uma métrica de qualidade entre 42 a 50 por cento, com exceção da última pessoa que deu várias voltas pelo mapa e obteve 39%. Estes valores foram bem condizentes com os obtidos pelo BOT, utilizando 10 jogos, com a qualidade de $42,36\%$.

Um dos motivos para os baixos valores de qualidade é dado pelo fato dos mapas gerados pelo algoritmo procedural não estarem adaptados ou totalmente uniformes e portanto a sua qualidade varia muito de acordo com as esperanças das métricas registradas. Mas este teste mostra que os valores obtidos pelos jogos automatizados são próximos aos reais e podem servir de uma primeira base para a criação de adaptação de mapas.

Foi solicitado aos usuários que elencassem o mapa preferido entre os 5 jogadores, sendo que 4 dos 5 usuários escolheram o mapa gerado manualmente. Apesar do mapa manual ter sido construído com regras similares ao procedural (somente salas quadradas), possivelmente a disposição de itens e inimigos do mapa influenciou a escolha dos usuários. Vale reforçar que o algoritmo procedural utilizado não está de forma alguma otimizado, pois este não é o objetivo do trabalho.

3.4 Ferramentas produzidas

A realização do trabalho resultou três aplicações que podem servir para diversos propósitos para auxiliar a produção mais efetivas de jogos do gênero *roguelike*.

A ferramenta principal, nomeada *MetaEngine*, é capaz de realizar simulações de jogos seguindo as regras básicas do gênero *roguelike*. Com a extração de métricas de seus jogos, ela computa um coeficiente de qualidade para validar se um dado mapa segue as expectativas. As simulações podem ser executadas manualmente através de um jogador humano ou automaticamente com uma inteligência artificial para o controle. O usuário da ferramenta pode atribuir o que ele deseja que seja o seu valor esperado, e através da análise dos resultados de vários jogos obter uma métrica de qualidade. A ferramenta principal pode receber como entrada um algoritmo procedural e gerar o mapa a ser testado em tempo de execução do programa ou utilizar um mapa carregado de um arquivo.

A métrica de qualidade geral do mapa pode em alguns casos não se tornar muito eficiente caso o mapa conste de muitos pontos de convergência, o que poderá causar a um declive na qualidade. Porém, uma vez extraídos os resultados de um mapa, é possível a análise dos valores obtidos através dos gráficos para dar ao *game designer* melhor idéia de como está a disposição do seu mapa de forma rápida, o que em si já é um resultado do trabalho.

A segunda ferramenta, nomeada *MapBuilder*, é a ferramenta auxiliar de construção de mapas. Enquanto o objetivo original desta ferramenta foi a construção de mapas manuais para serem testados contra os procedurais, ela foi evoluindo e se tornou uma ótima interface para se testar algoritmos procedurais feitos em lua com a sua mini-IDE para a edição e testes em tempo de execução de *scripts* Lua. Ou seja, a ferramenta auxiliar poderia ser usada a parte para a produção algoritmos procedurais com visualização rápida e eficiente.

Por ultimo, não precisamente uma ferramenta, mas um pequeno programa em Python para a visualização das métricas extraídas que pode ser utilizado para exibir os resultados extraídos da inteligência artificial graficamente, caso o *game designer* deseje estudar os valores das métricas obtidas pelas simulações para realizar sua própria análise.

A inteligência artificial, embora o foco do projeto não tenha sido o seu desenvolvimento, pode ser parametrizada para focar prioritariamente em itens, novos blocos inexplorados, ou um bloco solitário esquecido. Enquanto isto está longe do ideal para simular o comportamento e modos de jogabilidade de seres humanos, é possível obter um nível de variação para os testes realizados. O algoritmo utilizado foi baseado e adaptado da exploração automática de *TOME 4* (DARKGOD, 2013), um *roguelike* de código aberto.

4 Conclusão

O trabalho mostrou que é possível a criação de ferramentas automatizadas de teste para mapas e jogos procedurais. Embora diversos fatores ainda estejam afetando negativamente as métricas, como a incapacidade de simular perfeitamente o comportamento humano, a abordagem utilizada no trabalho pode ser utilizada para obter uma métrica de qualidade de forma rápida e sem a interação humana, que pode ser utilizada como *feedback* sobre o estado de um mapa gerado ou desenvolvido durante um jogo simulado.

O experimento conduzido com usuários mostrou qualidades obtidas semelhantes com a obtida pela inteligência artificial, o que reforça que utilização deste método de testes pode servir de auxílio no desenvolvimento de mapas

O objetivo deste trabalho nunca foi retirar o usuário do processo, e sim deixar nas mãos do *game designer* uma poderosa ferramenta, para que o processo de seleção de mapas ou de criação de algoritmos procedurais sejam mais rápidos e efetivos. Deste ponto de vista, os resultados obtidos deste projeto demonstram que a capacidade de extrair e analisar métricas por simulações automatizadas condizem com a realidade e podem acelerar a forma como são produzidos e testados mapas, sejam eles procedurais ou manuais.

Alguns pontos do trabalho como a inteligência artificial e os mapas procedurais testados foram limitados pelo escopo e podem ser melhorados como trabalhos futuros, dando a ferramenta principal métricas mais próximas a casos reais.

Referências

ADAMS, D. Automatic generation of dungeons for computer games. The University of Sheffield, 2002. Citado 3 vezes nas páginas 20, 21 e 22.

CHUNSOFT. *Chunsoft - Pokemon Mistery Dungeon Page (jp)*. 2013. Disponível em: <http://www.spike-chunsoft.co.jp/games/pokedun/>. Citado na página 15.

CHUNSOFT. *Izuna Legend of the Unemployed Ninja*. 2013. Disponível em: <http://www.atlus.com/izuna/>. Citado na página 15.

DARKGOD. *Tales Of Maj'Eyal*. 2013. Disponível em: <http://te4.org>. Citado 2 vezes nas páginas 33 e 56.

DISCOE, B. *Artificial Terrain Generation*. 2013. Citado na página 20.

DOULL, A. *Procedural Generation Content Wiki*. 2013. Disponível em: <http://pcg.wikidot.com/>. Citado na página 17.

ELLIPTIC. *Dungeon Crawl Stone Soup - Contest Results*. 2013. Disponível em: <http://crawl.develz.org/wordpress/0-13-tournament-results>. Citado na página 40.

FARNELL, A. Designing sound. 2010. Citado 2 vezes nas páginas 20 e 21.

FARNELL, A. *Procedural computational audio. Palestra, parte 3*. 2013. Disponível em: <http://www.youtube.com/watch?v=qUfOKBMlcuY>. Citado na página 21.

FRY, R. *Let's Play Angband*. 2009. Disponível em: <http://www.youtube.com/watch?v=TvZhkRkEoZc>. Citado na página 33.

GOKTAS, G. *Controlled Chaos - Procedural Content Generation*. 2013. Disponível em: <http://www.youtube.com/watch?v=fZPyj-53lSU>. Citado na página 20.

HELY, T. *How to Use BSP Trees to Generate Game Maps*. 2013. Disponível em: <http://gamedevelopment.tutsplus.com/tutorials/how-to-use-bsp-trees-to-generate-game-maps--gamedev-12268>. Citado na página 52.

HENDRIKX SEBASTIAAN MEIJER, J. V. D. V. A. I. M. Procedural content generation for games: A survey. Delft University of Technology, the Netherlands, 2011. Disponível em: <http://www.st.ewi.tudelft.nl/~iosup/pcg-g-survey11tomccap_rev_sub.pdf>. Citado na página 15.

JEF, O. F. *Let's Play Caves of Qud 01*. 2011. Disponível em: <http://www.youtube.com/watch?v=jKk9eSTLisQ>. Citado na página 33.

JULIAN, T. et al. Search-based procedural content generation: A taxonomy and survey. IT University of Copenhagen, University of Central Florida and Imperial College London, 2011. Citado na página 18.

KOENEKE, R. A. *Moria - Wikipedia*. 2013. Disponível em: <http://roguebasin.roguelikedevelopment.org/index.php?title=Moria>. Citado na página 24.

MONTGOMERY, D.; RUNGER, G. *Applied Statistics and Probability For Engineers*. [S.l.: s.n.], 2003. 98-116 p. Citado na página 43.

MOTT, T. (Ed.). *1001 Videogames para jogar antes de morrer*. [S.l.]: Editora Sextante, 2013. Citado na página 24.

NAUGHTYYT. *Graph Grammar based Procedural Generation for a Roguelike*. 2013. Disponível em: <http://www.youtube.com/watch?v=RAtdFKiqs34>. Citado 2 vezes nas páginas 20 e 22.

PENTON, R. (Ed.). *Data Structures for Game Programmers*. [S.l.]: Muska & Lipman/Premier-Trade; 1 edition (November 25, 2002), 2002. Citado 2 vezes nas páginas 34 e 69.

ROGUETEMPLE. *Roguelike Definition 2.0*. 2013. Disponível em: <http://www.roguetemple.com/roguelike-definition/>. Citado na página 24.

THEUBERHUNTER. *Let's Play Caves of Qud(P1)*. 2012. Disponível em: <http://www.youtube.com/watch?v=OAfxXY-mZR8>. Citado na página 33.

TOLKIEN, J. R. R. (Ed.). *O Senhor dos Anéis - Edição Completa*. [S.l.]: MARTINS FONTES, 2001. Citado na página 24.

TOMPSON, J. *PROCEDURAL GENERATION - THE CAVES*. 2011. Disponível em: <http://noelberry.ca/2011/04/procedural-generation-the-caves>. Citado 2 vezes nas páginas 20 e 22.

TULLEKEN, H. *How to Use Perlin Noise in Your Games*. 2009. Disponível em: <http://devmag.org.za/2009/04/25/perlin-noise/>. Citado na página 20.

Apêndices

APÊNDICE A – Propagação dos turnos

A sucessão dos turnos se baseia em um atributo de custo de velocidade. Cada movimento do jogador irá causar no sistema a passagem de t unidades de tempo, sendo este número igual a quantidade de custo que o jogador necessita para realizar um movimento (turno). Inimigos mais comumente possuirão um atributo de custo para se moverem maior que o jogador, tornando assim possível um jogador lutar contra diversos inimigos mais fracos com chances reais de vitória.

Para o caso do jogador se mover e o inimigo não atingir o seu valor de custo para movimentação, este valor é armazenado e utilizado para o próximo turno, decrementando somente a quantidade utilizada ao se mover. Por exemplo: Um jogador com custo de movimento 100 se move e é visto por um inimigo de custo 150. O inimigo não se moverá, porém terá 100 de custo armazenado. O jogador realiza outro passo, deixando-o agora com 200 armazenado, o qual realiza o seu movimento de custo 150 e deixa armazenado 50 para o próximo turno, no qual conseguirá novamente se mover, sobrando zero e repetindo este processo (Figura 7 - A).

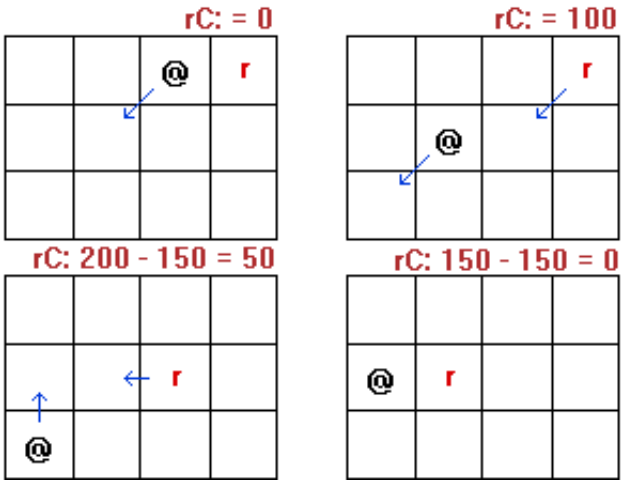

Figura 27 – Desenvolvimento de Turnos

APÊNDICE B – Leitura dos mapas

B.1 Blocos

Os primeiros valores do arquivo representam a disposição do mapa em relação ao seu tamanho, posição inicial do jogador, tipos e gráficos dos blocos. A formatação segue o padrão:

```
P_x . P_y
map_X-map_Y
id:tipo  id:tipo ...
```

Os primeiros dois valores P_x e P_y representam a posição inicial do jogador (a qual deve ser um bloco passável),sendo separados por um ponto. Na linha seguinte será indicado o tamanho nas direções do mapa em X e Y, com ambos valores separados por um traço. As map_Y linhas seguintes, representarão os blocos. Cada linha terá map_X identificadores dispostos por um *id* e *tipo*, separados por dois-pontos.

Cada conjunto de $id : tipo$ representa o bloco na sua posição de acordo com sua linha/coluna. Um identificador de um bloco pode ser:

- 0 - Bloco não passável
- Maior que 1 - Bloco passável

O segundo identificador *tipo* é o representativo visual que o bloco terá dentro do jogo. O *tipo* nada mais é do que um índice de recorte utilizado de um arquivo carregado pela ferramenta encontrado em *"data/img/tileset.png"*. Este arquivo pode ser de qualquer tamanho, e o jogo irá utilizá-lo para extrair o seus blocos. A imagem será recortada em pedaços de 16x16 pixels e será atribuída cada pedaço a um índice. Desta forma um *tileset.png* de 160x32 pixels terá índices entre 0 e 19 para representações. Os índices crescem horizontalmente, e ao chegar ao final de uma linha vão para a linha de baixo.

B.2 Inimigos e Itens

Uma linha imediatamente após o último bloco ser descrito no arquivo *.map*, haverá então os itens e inimigos presentes no mapa. Ao contrário dos blocos, os inimigos e itens não precisam ser dispostos seqüencialmente de acordo com suas posições e podem estar intercalados entre si.

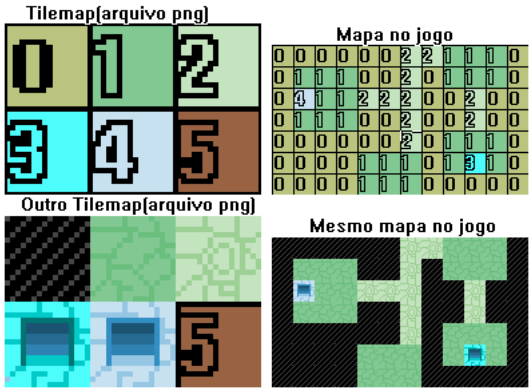

Figura 28 – Mapa, '3' e '4' indicando pontos de saída e entrada do mapa

Existem 3 variações possíveis de entradas nesta parte do formato. Elas são identificadas por uma linha contendo os seguintes textos:

- "*Item*:" - Itens de recuperação de vida ou de incrementos de atributos.
- "*Gold*:" - Dinheiro, para simulação dos benefícios do nível.
- "*Enemy*:" - Inimigos.

A linha subseqüente a um destes três textos conterá uma seqüencia de valores representando suas características. Em todos os três casos, os valores iniciam-se por dois identificados: x,y, representando as coordenadas do bloco x, y em que serão colocados.

B.2.1 *Gold* - Dinheiro

O mais simples deles, contém apenas mais uma variável g que representa a quantidade de dinheiro. O *sprite* escolhido para o dinheiro é automaticamente definido pelo sistema. Os primeiros seis *sprites* do mapa de itens representam os estados do dinheiro, que são definido como:

- 1-5g - *Sprite* 0
- 6-15g - *Sprite* 1
- 16-30g - *Sprite* 2
- 31-50g - *Sprite* 3
- 51-100g - *Sprite* 4
- 100g+ - *Sprite* 5

B.2.2 *Enemy* - Inimigo

Descrito pelos atributos: `hp`,`atk`,`def`,`range`,`cost`,`sprIDx`, `sprIDy`. Os primeiros valores são os atributos básicos do inimigo: sua vida, ataque e defesa.

O valor de *range* indica a visão do inimigo, isto é, a quantidade de blocos de distância que ele conseguirá observar o jogador e tomar a decisão de atacá-lo. O valor *cost* representa o custo de movimentação do inimigo, ou seja, quanto menor o valor, mais rápido será o inimigo. Por fim, os últimos dois valores representam a posição X e Y do arquivo gráfico para visualização do inimigo (*data/img/chars.png*).

B.2.3 Item

Este é o mais complicado dentre os três tipos de entrada do mapa, uma vez que pode abranger tipos de itens diferentes em uma mesma linha. É determinado pelos valores: `buff`, `hp`, `mp`, `atk`, `def`, `sprIDx`, `sprIDy`

O primeiro e mais importante parâmetro ($buff$) indica se o item é um item de recuperação ou de atributos. Tem valor 0 caso seja item de atributo ou 1 caso seja de recuperação.

O parâmetro *hp* e *mp* serão apenas utilizados caso o item possua o valor $buff$ igual a 1. E *atk* e def representam o quanto de ataque e defesa o jogador irá ganhar caso seja um item de aumento de atributos.

Similarmente aos inimigos, os últimos dois parâmetros indicam a posição no arquivo de imagem que irá representar graficamente o item em questão. Este arquivo é nomeado *data/img/itens.png*.

Um exemplo de um pequeno mapa 3x3 é dado a seguir (Figura 9 - B.2.3):

```
1.0
3-3
0:0 1:1 0:0
1:1 1:1 1:1
1:1 1:1 2:4
Gold:
0,1,10
Item:
1,2,1,5,0,0,0,1,2
Item:
0,1,0,0,0,1,0,0,1
Enemy:
0,2,5,2,1,3,200,2,0
```

Figura 29 – Representação do mapa - Objetos utilizados em destaque

APÊNDICE C – Algoritmo A*

O algoritmo $A*$ (PENTON, 2002) é utilizado para se achar um caminho entre dois pontos. Ele se baseia no conceito de exploração e descoberta de nós para se guiar e utiliza-se de heurísticas para determinar qual o melhor caminho de se expandir e evitar o desperdício de processamento, garantindo um provável caminho na direção certa. Ele utiliza uma fórmula de custo dado por:

$$F = G + H \tag{C.1}$$

onde F é o custo total para a realização da abertura do nó, G representa o custo para se mover em linha reta do ponto inicial ao ponto desejado e H representa o custo estimado para se mover de um determinado quadrado até o destino. O valor de H é usualmente chamado de heurística por ser apenas uma estimativa do valor real.

Desta forma, a cada quadrado explorado, o algoritmo irá analisar o custo de todos os seus nós descobertos, que são nós adjacentes aos explorados, e explorar o nó cujo o custo de F for o menor possível, repetindo este processo até que se encontre destino determinado.

APÊNDICE D – Distribuição t de student

Publicada por William Sealy Gosset, cujo pseudônimo escolhido foi *Student*, a distribuição t representa uma curva probabilística simétrica e companiforme, semelhante a curva normal padrão com caudas mais alargadas.

A fórmula para se calcular a distribuição t de Student pode ser dada por:

$$t = \frac{\mu - M}{\frac{s}{\sqrt{N}}} \tag{D.1}$$

sendo μ a média amostral, M a média populacional, s a variância amostral e N o número de amostras.

O teste de hipótese Student t é outro teste realizado muito na comparação entre dois eventos, podendo-se comparar duas amostras para se determinar se, por exemplo, seus médias populacionais são diferentes. Este processo é utilizado em situações para determinar se um dado novo método é mais efetivo que um antigo.

APÊNDICE E – Chi Quadrado

A função χ^2(Chi Quadrado) é um teste de hipótese que destina-se a encontrar o valor da dispersão para duas variáveis nominais, avaliando a associação dentre elas. Ao contrário de vários outros testes, o teste Chi Quadrado não é parametrizado, isto é, não depende de valores populacionais como média e variância.

A sua base se da na comparação entre a as proporções para observar as divergências entre as frequências observadas e esperadas para dado evento. Um dos usos deste teste de hipótese é a realização de um teste de aptidão para descobrir se um dado resultado de um evento se encaixa em um determinada função esperada.

Por exemplo, se um dado de 6 faces for jogado 30 vezes e seus resultados forem anotados em uma tabela. Pode-se utilizar o teste de hipótese Chi Quadrado para verificar se estes valores condizem com o esperado de um dado de 6 faces "justo", isto é, com chances iguais de obtenção dos valores.

A fórmula para se calcular pode ser dada por:

$$\chi^2 = \sum[\frac{(o-e)^2}{e}] \tag{E.1}$$

sendo 'o': valor obtido; 'e': valor esperado

APÊNDICE F – Scripts procedurais

F.1 Salas simples

```
1  -- Baseado em um video de Rachel Morris
2  -- http://www.youtube.com/watch?v=XlSlKwZit8g:
3
4  drawTime = 10
5  --Parametros possiveis
6  --    Numero de salas
7  --    Tamanho das salas
8  --    Largura e Altura das salas
9
10 n_Salas = 8
11 tamanho = 3
12 min_width = 2
13 max_width = 3
14 map_w = 40
15 map_h = 30
16
17 TILE_FLOOR = 3
18 TILE_CENTER = 2
19 TILE_END = 4
20 TILE_START = 2
21 TILE_CORRIDOR = 1
22 coolDebug = false
23
24 Sala = {}
25 Sala.__index = Sala
26
27 function Sala.create(x, y)
28    local sala = {}              -- our new object
29    setmetatable(sala,Sala)  -- make Account handle lookup
30
31    sala.x = x
32    sala.y = y
33    return sala
```

```
34 end
35
36 function isChance(chance)
37   mChance = chance/100;
38   if mChance > math.random() then
39     return true
40   else
41     return false
42   end
43 end
44
45 map:createMap(map_w, map_h, Tile.BLOCK, -1)
46
47 mSalas = {}
48
49 --Cria salas
50 for i = 1, n_Salas do
51   s = Sala.create( math.floor(math.random() * map_w), ↙
       ↳ math.floor(math.random() * map_h) )
52   mSalas[#mSalas+1] = s;
53   map:setTile(s.x,s.y,TILE_CENTER,Tile.PASS)
54   if coolDebug then sleep(drawTime) end
55 end
56
57 --Liga salas
58 for i = 1, #mSalas-1 do
59   --Origen
60   n_x = mSalas[i].x
61   n_y = mSalas[i].y
62   --Destino
63   d_x = mSalas[i+1].x
64   d_y = mSalas[i+1].y
65   --Cria caminho em X
66   while n_x ~= d_x do
67     if n_x < d_x then
68       n_x = n_x+1
69     elseif n_x > d_x then
70       n_x = n_x-1
71     end
```

```
72      if n_x < 0 or n_x >= map_w then
73        break
74      end
75      map:setTile(n_x, n_y, TILE_CORRIDOR, Tile.PASS)
76    end
77    if coolDebug then sleep(drawTime) end
78    --Cria caminho em Y
79    while n_y ~= d_y do
80      if n_y < d_y then
81        n_y = n_y+1
82      elseif n_y > d_y then
83        n_y = n_y-1
84      end
85      if n_y < 0 or n_y >= map_h then
86        break
87      end
88      map:setTile(n_x, n_y, TILE_CORRIDOR, Tile.PASS)
89      if isChance(20.0) then
90        goldCreated = math.floor(math.random() * 100)
91        res:addGold(n_x, n_y, goldCreated)
92      elseif isChance(8.0) then
93        res:addEntityByIndex(n_x, n_y, math.random(0,5))
94      elseif isChance(40.0) then
95        res:addItemByIndex(n_x, n_y, math.random(0,5))
96      end
97    end
98    if coolDebug then sleep(drawTime) end
99  end
100
101 --Enlarga salas
102 for i = 1, #mSalas do
103   left_w = min_width + math.floor(math.random() * max_width)
104   right_w = min_width + math.floor(math.random() * max_width)
105   up_w = min_width + math.floor(math.random() * max_width)
106   bottom_w = min_width + math.floor(math.random() * max_width)
107
108   startX = mSalas[i].x - left_w
109   for ix = 0, left_w+right_w do
110     if startX + ix < 0 then
```

```
111       --Continue
112     elseif startX + ix >= map_w then
113       break;
114     else
115       startY = mSalas[i].y-up_w
116       for iy = 0, up_w+bottom_w do
117         if startY+iy < 0 then
118           --Continue
119         elseif startY+iy >= map_h then
120           break
121         end
122         map:setTile(startX+ix, startY+iy, TILE_FLOOR, Tile.PASS)
123       end
124     end
125   end
126     if coolDebug then sleep(drawTime) end
127 end
128
129 for i = 2, #mSalas-1 do
130   map:setTile(mSalas[i].x, mSalas[i].y,TILE_CENTER, Tile.PASS)
131   if coolDebug then sleep(drawTime) end
132 end
133 map:setTile(mSalas[1].x, mSalas[1].y,TILE_START, Tile.START)
134 player:setPos(mSalas[1].x, mSalas[1].y)
135
136 if coolDebug then sleep(drawTime) end
137 map:setTile(mSalas[#mSalas].x, mSalas[#mSalas].y,TILE_END, Tile.END)
138 if coolDebug then sleep(drawTime) end
```

F.2 Salas com árvore de particionamento espacial binário

```
1 -- Baseado no artigo:
2 -- ↙
     ↳ http://gamedevelopment.tutsplus.com/tutorials/how-to-use-bsp-trees-t
3 MIN_LEAF_SIZE = 8
4 MAX_LEAF_SIZE = 18
5
6 drawTime = 10
7 --Parametros possiveis
8 --    Chance de dividir salas horizontal
```

```
9  --    Tamanho das salas internas
10
11 coolDebug = false
12
13 function isChance(chance)
14   mChance = chance/100;
15   if mChance > math.random() then
16     return true
17   else
18     return false
19   end
20 end
21
22 -- range(start)             returns an iterator from 1 to a (step = ↙
     ↳ 1)
23 -- range(start, stop)       returns an iterator from a to b (step = ↙
     ↳ 1)
24 -- range(start, stop, step) returns an iterator from a to b, ↙
     ↳ counting by step.
25 range =
26 function (i, to, inc)
27   if i == nil then return end -- range(--[[ no args ]]) -> return ↙
       ↳ "nothing" to fail the loop in the caller
28
29   if not to then
30     to = i
31     i  = to == 0 and 0 or (to > 0 and 1 or -1)
32   end
33   -- we don't have to do the to == 0 check
34   -- 0 -> 0 with any inc would never iterate
35   inc = inc or (i < to and 1 or -1)
36
37   -- step back (once) before we start
38   i = i - inc
39
40   return function () if i == to then return nil end i = i + inc ↙
       ↳ return i, i end
41 end
42
```

```
43 Leaf = {}
44 Leaf.__index = Leaf
45
46 function Leaf.create(x, y, width, height)
47    local leaf = {}              -- our new object
48    setmetatable(leaf,Leaf)  -- make Account handle lookup
49
50    leaf.x = x
51    leaf.y = y
52    leaf.width = width
53    leaf.height = height
54    return leaf
55 end
56
57 function Leaf:split()
58   -- Split leaf into 2 children
59   if (self.leftChild ~= null or self.rightChild ~= null) then
60     return false -- Already split!
61   end
62
63   --Determine direction of split
64   --If width >25% larger than height, vertical split
65   --If height >25% larger than width, horizontal
66   local splitH = (math.random() > 0.5)
67   if( self.width > self.height and (self.height / self.width) >= ↙
       ↳ 0.05) then
68     splitH = false
69   elseif( self.height > self.width and (self.width / self.height) ↙
       ↳ >= 0.05) then
70     splitH = true
71   end
72
73   local max
74   if splitH then
75     max = self.height
76   else
77     max = self.width
78   end
79
```

```
80    --Menor divis?o de leaf: 6
81    max = max-MIN_LEAF_SIZE
82
83    --Area muito pequena para dividir mais uma vez
84    if(max <= MIN_LEAF_SIZE) then
85      return false
86    end
87
88    local split = math.random(MIN_LEAF_SIZE, max)
89    if(splitH) then
90      self.leftChild = Leaf.create(self.x, self.y, self.width, split)
91      self.rightChild = Leaf.create(self.x, self.y + split, ↙
          ↳ self.width, self.height - split)
92    else
93      self.leftChild = Leaf.create(self.x, self.y, split, self.height)
94      self.rightChild = Leaf.create(self.x + split, self.y, ↙
          ↳ self.width - split, self.height)
95    end
96
97    return true --Splitted
98  end
99
100 function Leaf:draw(tileGfx, tileType)
101   if (tileGfx == null) then tileGfx = 1 end
102   if (tileType == null) then tileType = 1 end
103   for i=0, self.width-1 do
104     for j=0, self.height-1 do
105       map:setTile(self.x+i,self.y+j,tileGfx,tileType)
106     end
107   end
108
109   sleep(drawTime)
110 end
111 function Leaf:drawLines(tileGfx, tileType)
112   if (tileGfx == null) then tileGfx = 1 end
113   if (tileType == null) then tileType = 1 end
114   for i=0, self.width-1 do
115     for j=0, self.height-1 do
116       if i == 0 or j == 0 or i == self.width-1 or j == ↙
```

```
          ↪ self.height-1 then
117         map:setTile(self.x+i,self.y+j,tileGfx,tileType)
118       end
119     end
120   end
121   sleep(100)
122 end
123
124 function Leaf.x()
125   return self.x
126 end
127
128 function Leaf.y()
129   return self.y
130 end
131
132 function Leaf.width()
133   return self.width
134 end
135
136 function Leaf.height()
137   return self.height
138 end
139
140 function Leaf:createRooms()
141   --Gera salas e corredores
142   if( self.leftChild ~= null or self.rightChild ~= null) then
143     --Foi cortada, vai para filhas
144     if(self.leftChild ~= null) then
145       self.leftChild:createRooms()
146     end
147     if(self.rightChild ~= null) then
148       self.rightChild:createRooms()
149     end
150
151     if self.leftChild ~= null and self.rightChild ~= null then
152       self:createHall(self.leftChild:getRoom(), ↙
          ↪ self.rightChild:getRoom())
153     end
```

```
154   else
155     --Pronta para fazer a sala
156     local sizeW;
157     local sizeH;
158     local posX;
159     local posY;
160
161     --Sala pode ser do tamanho 3x3 at? o tamanho da left-2
162     local sizeW = math.random(3, self.width-2);
163     local sizeH = math.random(3, self.height-2);
164
165     --Coloca a sela dentro da leaf mas n?o encostado, para n?o ↙
          ↳ mergir salas
166     local posX  = math.random(1, self.width - sizeW-1);
167     local posY  = math.random(1, self.height - sizeH-1);
168
169     if coolDebug == true then
170       self:drawLines(4)
171     end
172     self.room = Leaf.create(self.x+posX, self.y+posY, sizeW, sizeH)
173     self.room:draw()
174
175     n_x = self.x+posX+(sizeW/2.0)
176     n_y = self.y+posY+(sizeH/2.0)
177
178     for n_x in range(self.x+posX, self.x+posX+sizeW-1) do
179       for n_y in range(self.y+posY, self.y+posY+sizeH-1) do
180         --Chance de criar objetos por tile da sala
181         if isChance(1.0) then
182           goldCreated = math.floor(math.random() * 100)
183           res:addGold(n_x, n_y, goldCreated)
184         elseif isChance(2.0) then
185           res:addEntityByIndex(n_x, n_y, math.random(0,5))
186         elseif isChance(4.0) then
187           res:addItemByIndex(n_x, n_y, math.random(0,5))
188         end
189
190
191       end
```

```
192     end
193
194
195
196   end
197 end
198
199 function Leaf:getRoom()
200   --Itera todos as leafs para achar a sala
201   if self.room ~= null then
202     return self.room
203   else
204     local lRoom
205     local rRoom
206     if self.leftChild ~= null then
207       lRoom = self.leftChild:getRoom()
208     end
209     if self.rightChild ~= null then
210       rRoom = self.rightChild:getRoom()
211     end
212
213     --Retorna o leaf com a sala , caso ambos tenham sala, retorna ↙
          ↳ aleat?rio um deles
214     if lRoom == null and rRoom == null then
215       return null
216     elseif rRoom == null then
217       return lRoom
218     elseif lRoom == null then
219       return rRoom
220     elseif math.random() > 0.5 then
221       return lRoom
222     else
223       return rRoom
224     end
225
226   end
227 end
228
229 function Leaf:createHall(l, r)
```

```
230   --Conecta as duas salas
231   local point1X =  math.random(l.x + 1, l.x+l.width-2)
232   local point1Y =  math.random(l.y + 1, l.y+l.height-2)
233   local point2X =  math.random(r.x + 1, r.x+r.width-2)
234   local point2Y =  math.random(r.y + 1, r.y+r.height-2)
235
236   local w       = point2X - point1X
237   local h       = point2Y - point1Y
238
239   local halls = {}
240
241   if w < 0 then
242     if h < 0 then
243       --Alterna caminhos , ex: esquerda cima,  cima esquerda
244       if math.random() > 0.5 then
245         halls[#halls+1] = Leaf.create(point2X, point1Y, ↙
              ↳ math.abs(w),1)
246         halls[#halls+1] = Leaf.create(point2X, point2Y, 1, ↙
              ↳ math.abs(h))
247       else
248         halls[#halls+1] = Leaf.create(point2X, point2Y, ↙
              ↳ math.abs(w),1)
249         halls[#halls+1] = Leaf.create(point1X, point2Y, 1, ↙
              ↳ math.abs(h))
250       end
251     elseif h > 0 then
252       --Alterna caminhos , ex: esquerda cima,  cima esquerda
253       if math.random() > 0.5 then
254         halls[#halls+1] = Leaf.create(point2X, point1Y, ↙
              ↳ math.abs(w),1)
255         halls[#halls+1] = Leaf.create(point2X, point1Y, 1, ↙
              ↳ math.abs(h))
256       else
257         halls[#halls+1] = Leaf.create(point2X, point2Y, ↙
              ↳ math.abs(w),1)
258         halls[#halls+1] = Leaf.create(point1X, point1Y, 1, ↙
              ↳ math.abs(h))
259       end
260     else -- if h == 0
```

```
261         halls[#halls+1] = Leaf.create(point2X, point2Y, ↙
                ↳ math.abs(w), 1)
262       end
263     elseif w > 0 then
264       if h < 0 then
265         --Alterna caminhos , ex: esquerda cima,  cima esquerda
266         if math.random() > 0.5 then
267           halls[#halls+1] = Leaf.create(point1X, point2Y, ↙
                ↳ math.abs(w),1)
268           halls[#halls+1] = Leaf.create(point1X, point2Y, 1, ↙
                ↳ math.abs(h))
269         else
270           halls[#halls+1] = Leaf.create(point1X, point1Y, ↙
                ↳ math.abs(w),1)
271           halls[#halls+1] = Leaf.create(point2X, point2Y, 1, ↙
                ↳ math.abs(h))
272         end
273       elseif h > 0 then
274         --Alterna caminhos , ex: esquerda cima,  cima esquerda
275         if math.random() > 0.5 then
276           halls[#halls+1] = Leaf.create(point1X, point1Y, ↙
                ↳ math.abs(w),1)
277           halls[#halls+1] = Leaf.create(point2X, point1Y, 1, ↙
                ↳ math.abs(h))
278         else
279           halls[#halls+1] = Leaf.create(point1X, point2Y, ↙
                ↳ math.abs(w),1)
280           halls[#halls+1] = Leaf.create(point1X, point1Y, 1, ↙
                ↳ math.abs(h))
281         end
282       else -- if h == 0
283           halls[#halls+1] = Leaf.create(point1X, point1Y, ↙
                ↳ math.abs(w), 1)
284       end
285     else -- if w == 0
286       if h < 0 then
287         halls[#halls+1] = Leaf.create(point2X, point2Y, 1, math.abs(h))
288       elseif h > 0 then
289         halls[#halls+1] = Leaf.create(point1X, point1Y, 1, math.abs(h))
```

```
290     end
291   end
292
293   for i, hall in ipairs(halls) do
294     hall:draw()
295   end
296
297 end
298
299 --Create map
300 map:createMap(30,30, Tile.BLOCK, -1)
301
302 -- Create leafs
303 local _leafs = {}
304 local root = Leaf.create(0,0,map.w,map.h)
305 _leafs[1] = root
306
307
308
309 local did_split = true
310 while did_split do
311   did_split = false
312
313   --map:setTile(0,1,1,1)
314   --map:setTile(0+i,0,1,1)
315   for i, l in ipairs(_leafs) do
316     if(l.leftChild == null and l.rightChild == null) then
317       --Se leaf muito grande ou 75% chance
318       if(l.width > MAX_LEAF_SIZE or l.height > MAX_LEAF_SIZE or ↙
             ↳ math.random() > 0.25) then
319         if(l:split()) then
320           _leafs[#_leafs+1] = l.leftChild
321           _leafs[#_leafs+1] = l.rightChild
322           did_split = true
323         end
324       end
325     end
326   end
327 end
```

```
328
329 root:createRooms()
330
331 left = root
332 right = root
333
334 while left.leftChild ~= null do
335   left = left.leftChild
336 end
337 map:setTile(left.x+(left.width/2.0), ↙
      ↳ left.y+(left.height/2.0),4,Tile.START)
338 player:setPos(left.x+(left.width/2.0), left.y+(left.height/2.0))
339
340 while right.rightChild ~= null do
341   right = right.rightChild
342 end
343 map:setTile(right.x+(right.width/2.0), ↙
      ↳ right.y+(right.height/2.0),4,Tile.END)
344 --root:draw()
345 -- create and use an Leaf
346 --leaf = Leaf.create(0,0,13,13)
347
348 --if leaf:split() then
349 --  map:createMap(10,10);
350 --else
351 --  map:createMap(1,1);
352 --end
```